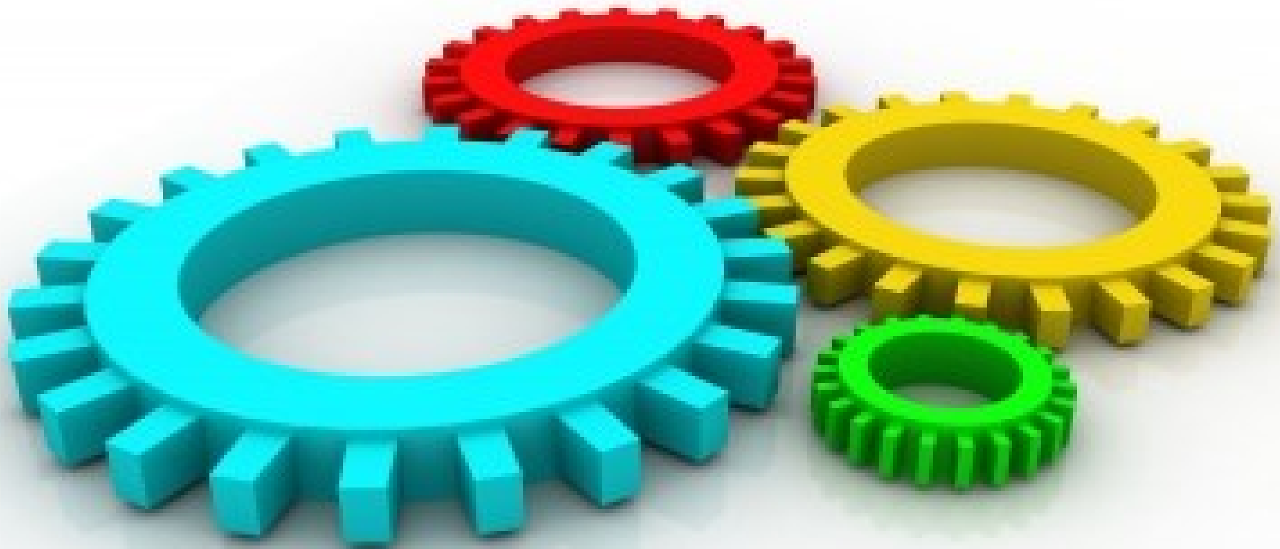
Connection Group Fun with App-V 5

Examples of Solving Issues with Connection Groups



Tim Mangan

TMurgent Technologies, LLP

January, 2014

Copyright © 2014 by TMurgent Technologies

Published as a service to the App-V Community by TMurgent Technologies.

No part of this work may be reproduced without permission of the author.

Cover image attribution: Creative Commons by jccreationzs

CONTENTS

1 Introduction 1
2 Living without the Group 2
3 Think like the Client, not the Sequencer 4
4 Obstructing the View 6
5 Fool the Redirection 8
6 Closing 10

1 INTRODUCTION

Connection Groups are the App-V 5 equivalent of *Dynamic Suite Composition* (DSC) in App-V 4, which is a way to virtualize two or more virtual application packages and have them work together dynamically as a single virtual application at the client.

Due to how App-V 5 is built, this can lead to having a solution to issues unsolvable by DSC in some cases, while creating new challenges to other combinations that previously worked well together. The purpose of this paper is to document some ideas (approaches) that have worked to solve a few of these Connection Group packaging issues. More important than the technique to solve one issue, these techniques should give practitioners ideas on how to approach many of the new challenges when working with Connection Group.

A summary of these ideas are given here:

- There are some situations where DSC was required in the past and Connection Groups are not needed in App-V 5.
- In some cases, you have to think backwards to build the group in App-V 5. Rather than think about how the App natively works when building a package that is dependent on another, you must think like the App-V client.
- There are times when masking items is required. This can, in some cases, be more difficult when Connection Groups are involved.
- Sometimes the limitation of writing to VFS files must be overcome. While this can be a problem in a single app, it tends to be more of a problem when independent apps are grouped.

Intended Audience

This paper assumes a relatively technical reader that has a fair amount of prior knowledge about how App-V 5 works, and in how Connections Groups perform.

It does not attempt to explain things to someone not well trained to package virtual applications in App-V.

2 Living without the Group

App-V 5 is less opaque than the prior versions, (some might call the old versions "overly virtualized"), allowing virtualized applications more ways to locate each other in scenarios where sharing of virtual environments was only required to enable one to launch the other.

Additionally, the new extensions in App-V 5 add new ways for apps to find each other that were blocked in App-V 4.

If the apps do not really share virtual components and any necessary data files live outside the virtual environment, sometimes you can get away with removing the grouping.

This tends to be useful only when the two packages are genuinely independent, with one of the packages starting an exe in the other package through an extension point or just launching it using the path variable.

Once in a while, these DSC combinations that were needed in App-V 4.* can be solved by just sequencing each package independently and testing.

In one situation, all that was needed was to add an "APP PATH" for the executable in the package that must call the second package. To make this work, the App Path would point to the client location where this would be found:

Package A includes an executable named "bar.exe". The file is installed to a sub-folder named "Floc" under the Primary Virtualization Application Directory (PVAD).

Package B includes an executable named "foo.exe". Foo needs to launch bar.exe, if present, when the user invokes some application feature. Foo passes a file from the user's documents area as part of the command line, but foo does not require any virtualized components of the bar package.

When packaged separately, foo cannot locate bar. Using the new App Paths extension point, we can add an AppPath to the foo package. The value name would be "foo.exe" and the value data would be the path to the folder containing bar.exe as the client would place it, when present. This typically would be "C:\ProgramData\App-V\{Package A ID GUID}\{Package A Version ID GUID}\root\Floc".

While this solves the problem without harming Package B when Package A is not present, it does create a dependency inside Package B to the version GUID of Package A that is not obvious and must be well documented. Fortunately the linkage is exposed and modifiable in the Package B DynamicConfig XML file without reopening the package

Of course another form of living without the connection group still exists and should not be forgotten – just create a single sequence of both applications. While sometimes this is not possible, if you are struggling with getting a Connection Group to work you should consider this approach.

3 Think like the Client, not the Sequencer

When creating DSC groups for a package with a dependency on another package, it is necessary to think like the sequencer works. This includes situations of sequencing a plug-in that is dependent on a main application, or a package that uses another package as middleware (like a Java package).

The technique used to generate the second package involves creating a sequencer environment that mimics the state of the sequencer when the first app was sequenced. Originally, this meant doing a native install of the dependent app on the sequencer the same way it was sequenced. Later, Microsoft added the ability to use the sequencer to expand the dependent package for us, laying it down the way it was sequenced. Often, this approach will work fine with App-V 5 Connection Groups. But not always.

Most applications that actively support plug-ins locate the plug-ins one of two ways. Either the plug-in drops a dll in a specified location or it adds a registry item under a specified key that contains the path to the dll.

Those that use the registry method seem to work pretty well. When the file/folder method is used, this can be a problem depending on how the application locates that folder. Most of the time, the application will determine where the folder is located by first determining where it was installed from the registry, and adding a folder offset to that path. When the dependent app was installed in its PVAD folder, this should work. It will see the PVAD folder name, and plug-in package overlays on top of that using a VFS path.

In one case, the app used the folder method, but located the folder by applying the relative path to the current working directory. At the client, this is not the PVAD folder where installed (which ultimately is a junction point) but instead the app will see a current working directory of something like C:\ProgramData\App-v\{GUID}\{GUID}\root. If the plug-in was created by expanding the dependent package, the plug-in files will not be seen in the expected location. Attempts to solve this by VFSing everything also fail.

The solution lies in creating the plug-in package in an environment that mimics the client environment instead of the sequencer.

After creating the dependent package, the sequencer is reverted and the package is natively installed to C:\ProgramData\App-V\{GUID}\{GUID}\root.

Then the plug-in is sequenced using a C:\PluginName PVAD with the plug-in files added where the native installed dependent package expects them. These files are therefore VFS'd to overlay exactly where the executable at the client will look for them.

This is the solution I used to get Connection Groups to work for plug-ins to Paint.Net. Once again, the plug-ins have a dependency of the dependent package id and package version, but here there is a connection group that should help to identify the dependency via management consoles.

It is also possible to use the Package GUID publishing junction point when performing the dependency native installation prior to sequencing the plug-in. This requires prior knowledge of how the dependent package will be published, as this junction point is in a different location when published globally or to the user. If this is well known, using this junction point would eliminate the dependency on the dependent package Version ID GUID allowing the dependent package to be independently upgraded without breaking the plug-in packages.

Ideally, it might be nice is the sequencer had an option to expand to local system like the client (however, this should NOT be a replacement of the current expand to local system).

4 OBSTRUCTING THE VIEW

There are times when it is necessary to obstruct the view of either real or potential data objects (folders, files, registry keys, or registry items) that appear in a lower layer. We run into this more often with a single package than in Connection Groups, but it turns out that the packages where this is needed are often part of a connection group anyway.

Often, what we want is to block the virtual application from seeing something that might be present in another form on the native system, but the technique is equally effective to block one member of a connection group from seeing another member.

There are two forms of blocking supported by an App-V package:

- One used when a package includes files or registry items under a given folder or key and that folder or key might exist in in a lower layer package or the native system.
- One used when a package does not contain a replacement but wants to hide the existence of a potential file, folder, registry key, or registry item.

The first form is usually automatically taken care of by the sequencer. When creating the package, the sequencer marks a custom property on Folders and Registry keys depending on how that Folder/Key came to be captured as part of the package. If the Folder or Key was created new during the monitoring, the item property is set to “Override Local” and if was indirectly captured because the Folder/Key already existed on the system but something below it in the file system or registry system was captured and it is coming along for the ride the item property is set to “Merge with Local”. Inside the sequencer editor (the multi-tabbed interface) you can see this settings by right clicking on the folder/key. This is sometimes referred to as the Transparency (or Pellucidity) of the item and you can change the property setting in the sequence editor.

The logic of the default behavior of the sequencer is that normally you want the package to see sub-items of a folder/key for normally present system sub-items. So if the package indirectly captures the Windows\System32 folder because a new dll was added in the package, the folder is marked “Merge with Local” so that the virtual application can also see and load additional dlls from that folder that are present on the end-user system. But if the app creates a new folder (such as C:\Program Files\Common Files\VendorAndAppName) we usually want to hide any presence of a different version of that app that might be locally installed.

The second form of blocking is not (currently) visible in the sequencer, so many people are less familiar with it. But while the sequencer is monitoring for changes, if a pre-existing file or registry system item is removed, a placeholder is added to the package with a custom deletion marker property. At the client, this is a indication that should the item be actually present at the client, it should not be visible to the package.

A great example of using blocking via deletion markers is sequencing a particular version of Java. Sometimes applications require one particular version of Java, but Java was designed to automatically locate and use the latest version installed in most cases. This is done by detecting the presence of additional registry sub- keys of the newer version.

So we might sequence version X of Java (or make a connection group containing a package with version X) and the client might have version X+1 natively installed, so the application would end up loading version X+1.

Aaron Parker wrote up a technique (see http://stealthpuppy.com/juggling-sun-java-runtimes-in-app-v/) to hide all other known versions of Java as part of your Java package. Unfortunately your Java package may need to be re-created when a newer version of Java becomes available to add the additional deletion marker, but the technique is well described.

The technique involved pre-creating the registry keys we want to block prior to starting the sequencer, and then deleting those keys as part of the Java package. So even if those versions are present at the client, they will not be seen.

So far in this chapter, I have described the behavior of one package and the client. It turns out that the situation gets far more complicated when you involve connection groups. When connection groups are involved, the client behavior is sometimes different depending on whether the situation involves the file system or registry system.[1] Most of the time, these differences are not important, however if when you use pellucidity or deletion markers in a connection group and do not get the behavior you expected, you should consult a paper that I previously wrote called “Pellucidity and Deletion Objects” (http://www.tmurgent.com/appv/index.php/resources/research/172-pellucidity-and-deletion-objects-in-app-v-5). In this paper I publish the results of a long series of tests detailing the results of all possible combinations of the local client and a connection group consisting of three packages. If your situation runs afoul due to the combination not doing what you expect, it should be possible to move the marker to the first or last package in the group (potentially a new dummy package consisting of only the marker).

[1] I cannot find a good reason for the differing behavior and attribute the difference to different development teams making separate choices on how to handle unusual potential situations that probably rarely occur.

5 FOOL THE REDIRECTION

One of the issues we have that is new to App-V 5, whether you are using a connection group or not, is dealing with apps that need to write files outside of the AppData area. Unlike App-V 4.*, at the client the virtual application in App-V 5 cannot write to locations that are in the VFS map of the package. This behavior is *almost* well defined in the Microsoft *App-V 5 SP2 Application Publishing and Client Interaction* guide[2]. Many times, we can solve this new issue by getting this by making this location be within the PVAD folder.

In one case, an app (which did not require a connection group) needed to overwrite a file that the installer always placed in C:\ProgramData\VendorName\Subfolder.

This app was solved by using that folder as the PVAD. Because this was a simple app and it only needed to write to this one location, it wasn't important where we actually installed the app to inside the sequencer. Typically I would probably install it under a different subfolder under the PVAD to keep within Microsoft guidance.

When you bring connection groups into the picture, if the two apps both have this kind of behavior, in two different "fixed" folder locations, such a fix might not work. The same would be true if a single app package had multiple "fixed" locations to installing to a subfolder of the PVAD can't keep at least one of the locations from being in the VFS.

To solve this, we can add our own Junction Points inside the package. Yes, junction points do virtualize!

To make this simple to explain, I will stick with the single package scenario, but once you understand this you can expand it to the multi-package connection group situation as well.

[2] http://www.microsoft.com/en-us/download/details.aspx?id=41635. This guide provides good examples of Copy-On-Write behavior and is well worth reading; unfortunately it is not a complete description of the COW behavior in all cases. You may want to search the download center for the latest version (leaving out the SP2 part of the title) in case the product interaction behavior changes in the future and/or is better described.

A large application had needs to write to two different fixed folder locations. After setting the PVAD to one of these fixed folder locations, we still had VFS write issues with the second location. Here is how that was solved.

The PVAD was set to the first location and the product installed to a subfolder of the PVAD. After installing the product, and still in monitoring mode, the second fixed folder was ***moved*** *to an additional subfolder of the PVAD. Then a directory junction point was created in the location of the original fixed folder location pointing to this new location (using the mklink command in a new command window).*

The Junction point was captured as a VFS location, but when the application tried to use a path starting with the second fixed path, the VFS brought in the junction point, which then redirected to the location under the PVAD. This in turn, hit the PVAD junction point that then referenced the redirected location under C:\ProgramData\App-V\{GUID}\{GUID}\root

When an application opens a file that passes through a junction point, the file handle opened is actually to the redirected location path, not the path the application requested. To the App-V client, this then looks like a PVAD file and allows the write (further redirecting the actual write to the Copy-On-Write area of the user's AppData.

The idea of using your own junction points to redirect file locations to different places can be used in many different ways. You could redirect a single file to be outside the bubble. If you are willing to set the Group Policy Object to change the OS default limitation of not following local to remote junction points, you could even place the folder or files up on a commonly accessible file share.

These ideas might not only solve rights issues, but could make it easy to update a package data file centrally rather than rev the package. Sometimes home-grown line of business apps have a data-file that updates on a frequent schedule (such as once a month) without changing the app itself, and expects the file to be local. If the app always opens the file for read-only, sequencing the junction point to redirect to a share location makes it easy to update the app without repackaging or performing any publishing. Care would be needed in swapping the file out on the share, and either an app maintenance period would be needed or a secondary junction point on the share could be used (if the GPO is set to allow remote-to-remote junction points) to avoid the maintenance window.

6 CLOSING

Of course, none of these techniques should be used unless necessary. Each has some drawbacks that need to be weighed against the benefit. Sometimes this will make sense to use these techniques, others maybe not.