This means, of course, that is premature to test PowerShell in Linux and expect any complete functionality. Nonetheless, I thought it important for us to "kick the tires" and look into specifically what's possible for Windows-to-Linux and Linux-to-Windows remoting within PowerShell.

Review the Windows environment

On the Windows side, I have a fully patched Windows Server 2016 workgroup machine. As you know, Windows Server 2016 and Windows 10 ship with PowerShell v5.1. Here, let me show you my $PSVersionTable results:

```
$PSVersionTable
Name                           Value
----                           -----
PSVersion                      5.1.14393.206
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.14393.206
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

Pay particular attention to the new PSEdition property--this gives us a quick sanity check as to which operating system we're attached to. For this tutorial, I decided to stay with PowerShell v5.1 instead of installing the latest alpha release (https://github.com/PowerShell/PowerShell/releases) from GitHub.

Install PowerShell on CentOS Linux

On the Linux side, I have a fully patched CentOS 7 system with the basic server packages installed. Microsoft has some PowerShell for Linux installation docs (https://blogs.msdn.microsoft.com/powershell/2017/02/01/installing-latest-powershell-core-6-0-release-on-linux-just-got-easier/); here I'll buzz through the simple workflow for CentOS Linux using their own code:

```
# Enter superuser mode
sudo su
# Register the Microsoft RedHat repository
curl https://packages.microsoft.com/config/rhel/7/prod.repo >
/etc/yum.repos.d/microsoft.repo
# Exit superuser mode
exit
# Install PowerShell and .NET Core
sudo yum install -y powershell
# Start PowerShell
powershell
```

The first thing we'll want to do after starting PowerShell on Linux is to check our version and PSEdition:

```
$PSVersionTable
Name                           Value
----                           -----
PSVersion                      6.0.0-alpha
PSEdition                      Core
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   3.0.0.0
GitCommitId                    v6.0.0-alpha.17
CLRVersion
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

PowerShell remoting architecture

As you know, Windows hosts can communicate using Windows Remote Management (WinRM)-based PowerShell remoting sessions. WinRM is an implementation of the Web Services for Management (WS-Man) specification, which in turn takes advantage of firewall-friendly HTTP (TCP port 5985) or HTTPS (TCP port 5986) protocols to establish the communications channel. This channel uses more-or-less "traditional" XML/SOAP web services.

Of course, the WinRM service is Windows-specific, so the PowerShell team has some work to configure the Linux PowerShell engine to do WS-Man remoting. The best I can make out (this situation is murky; I challenge you to read the PowerShell project's issues lists) is that Linux supports WS-Man remoting through a combination of PowerShell Remoting Protocol (https://msdn.microsoft.com/en-us/library/dd357801.aspx) (MS-PSRP) and an Open Management Infrastructure (OMI) provider (https://github.com/PowerShell/psl-omi-provider). Here is a nice deep-dive (http://www.hnwatcher.com/r/3632577/A-look-under-the-hood-at-Powershell-Remoting-through-a-cross-plaform-lens) into the relationship between WinRM and PSRP.

Whew--we are getting into the high conceptual weeds here! The bottom line is that we need to install both OMI and the OMI provider on our Linux box if we have any hope of accomplishing PowerShell remoting, much less Desired State Configuration (DSC).

If you don't know, OMI (https://github.com/Microsoft/omi) is a standard interface for exposing Common Information Model (CIM) data on non-Windows systems. We Windows systems administrators know CIM as the Windows Management Instrumentation (WMI) repository on Windows systems.

We don't have the space to do a step-by-step walkthrough, so your next configuration steps in Linux are to:

1. Install and configure OMI server (https://github.com/Microsoft/omihttps:/github.com/Microsoft/omi)
2. Install the PSRP Linux support library (https://github.com/PowerShell/psl-omi-provider)

Now that we've come this far, let's see what's possible.

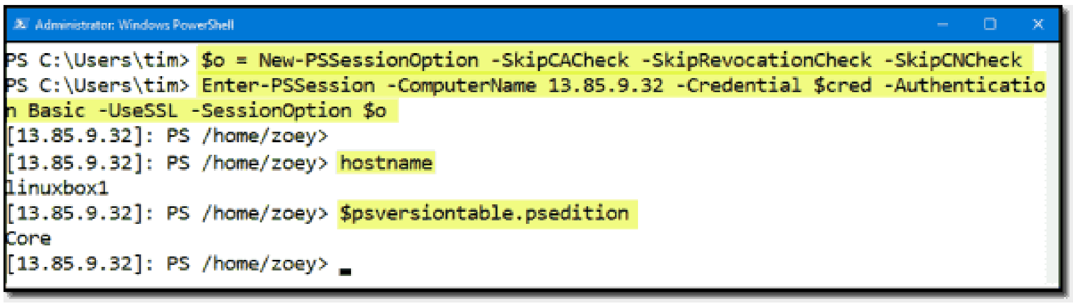
Windows-to-Linux remoting

The first thing I wanted to try was an interactive remoting session from Windows Server 2016 to CentOS Linux. Per the docs, this is what I did, and as you can see from the subsequent screenshot, I was successful:

```
# Store the appropriate Linux credential as an object
$cred = Get-Credential

# Disable cert checking (this is a dev/test environment)
$o = New-PSSessionOption -SkipCACheck -SkipRevocationCheck -SkipCNCheck

# Start the session, specifying our custom remoting configuration
Enter-PSSession -ComputerName 'linuxbox1' -Credential $cred -Authentication basic -UseSSL -SessionOption $o
```



```
PS C:\Users\tim> $o = New-PSSessionOption -SkipCACheck -SkipRevocationCheck -SkipCNCheck
PS C:\Users\tim> Enter-PSSession -ComputerName 13.85.9.32 -Credential $cred -Authenticatio
n Basic -UseSSL -SessionOption $o
[13.85.9.32]: PS /home/zoey>
[13.85.9.32]: PS /home/zoey> hostname
linuxbox1
[13.85.9.32]: PS /home/zoey> $psversiontable.psedition
Core
[13.85.9.32]: PS /home/zoey> _
```

(https://4sysops.com/wp-content/uploads/2017/03/Windows-to-Linux-PowerShell-remoting.png)

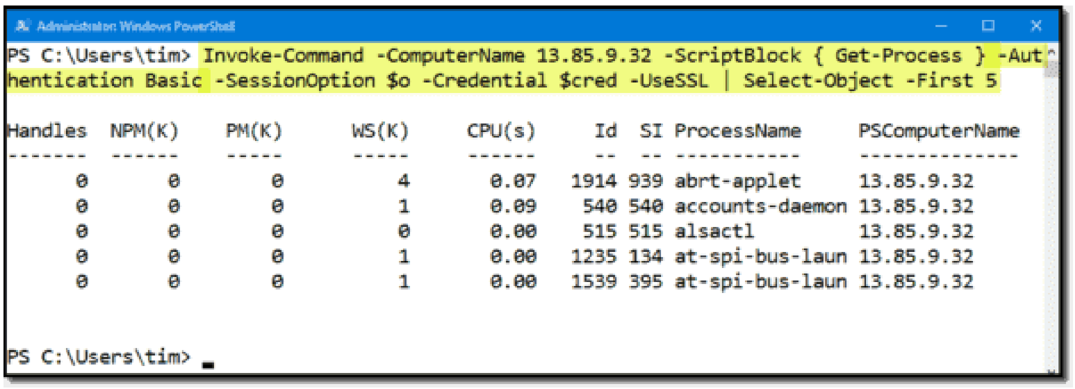
Windows to Linux PowerShell remoting

Cross-platform PowerShell remoting uses HTTP authentication methods, specifically either basic access or Windows NT LAN Manager (NTLM). For our purposes, we went with the easier, safer choice while the PowerShell engine is in alpha.

The second thing I wanted to do is to use *Invoke-Command* to run an ad-hoc script-block as well as a .ps1 script.

For this it worked as long as I reused the "hobbled" session configuration options. Remember--PowerShell for Linux and macOS is in alpha state, so we're just testing here.

Invoke-Command -ComputerName 13.85.9.32 -ScriptBlock { Get-Process } -Authentication Basic -SessionOption $o -Credential $cred -UseSSL | Select-Object -First 5



```
PS C:\Users\tim> Invoke-Command -ComputerName 13.85.9.32 -ScriptBlock { Get-Process } -Aut
hentication Basic -SessionOption $o -Credential $cred -UseSSL | Select-Object -First 5

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName      PSComputerName
-------  ------    -----      -----     ------     --  -- -----------      --------------
      0       0        0          4       0.07   1914 939 abrt-applet      13.85.9.32
      0       0        0          1       0.09    540 540 accounts-daemon 13.85.9.32
      0       0        0          0       0.00    515 515 alsactl          13.85.9.32
      0       0        0          1       0.00   1235 134 at-spi-bus-laun 13.85.9.32
      0       0        0          1       0.00   1539 395 at-spi-bus-laun 13.85.9.32


PS C:\Users\tim> _
```

(https://4sysops.com/wp-content/uploads/2017/03/Obtaining-a-process-listing-from-a-remote-Linux-server.png)

Obtaining a process listing from a remote Linux server

I have a simple **hello.ps1** script that contains a single Write-Output 'Hello world!' statement. Let's run it from Windows:

```
PS C:\Users\tim> Invoke-Command -ComputerName 13.85.9.32 -FilePath 'C:\hello.ps1' -Authentication Basic -SessionOption $o -Credential $cred -UseSSL
Hello world!
PS C:\Users\tim>
```

That worked! It's important for us to remember that PowerShell on Linux and macOS exists on top of the portable .NET Core runtime and not the full .NET Framework we have in Windows. This means you'll likely need to modify your PowerShell scripts a bit for them to work locally on Linux or macOS. On the other hand, you could use implicit remoting to load remote modules into the local system's runspace.

Linux-to-Windows remoting

The PowerShell development team is much further along in Windows-to-Linux PowerShell remoting than it is the other way around. Frankly, there are more moving parts in this approach, especially if you're using NTLM for authentication on the Windows side.

On your Windows test system, you can enable basic authentication and allow unencrypted connections by running the following commands:

```
winrm set winrm/config/Service/Auth @{Basic="true"}
winrm set winrm/config/Service @{AllowUnencrypted="true"}
```

And then in Linux we can do this to make an interactive session with the remote Windows server:

```
$cred = Get-Credential
Enter-PSSession -ComputerName 'winserver1' -Credential $cred -Authentication Basic
```

The PSRP repository (https://github.com/PowerShell/psl-omi-provider) at GitHub briefly explains the above procedure.

Upcoming: secure shell-based remoting

From what those who know have told me (https://powershell.org/2015/06/09/why-remoting-vs-ssh-isnt-even-a-thing/), PowerShell inventor Jeffrey Snover didn't want to use WS-Man for PowerShell remoting. Instead, his vision of cross-platform PowerShell remoting relied upon the longstanding Secure Shell (SSH) cryptographic network protocol.

In fact, the PowerShell team has been working on their Windows OpenSSH port (https://github.com/PowerShell/Win32-OpenSSH) for a long time, and the repository is still very active. At this point I'd say that the team's PowerShell Remoting Over SSH (https://github.com/PowerShell/PowerShell/blob/866b558771a20cca3daa47f300e830b31a24ee95/docs/new-features/remoting-over-ssh/README.md) article is the definitive general overview.

WS-Man and SSH do not represent an "either/or" proposition. You may find that using WS-Man is appropriate for some situations, and using SSH is appropriate for others. You'll be able to choose them at your convenience!