Administrator’s Guide to Windows PowerShell Remoting

Dr. Tobias Weltner
Aleksandar Nikolic
Richard Giles
This paper explains how to set up and run Windows PowerShell Remoting which is a new feature in Windows PowerShell 2.0 and allows you to run Windows PowerShell commands and scripts remotely. So, before moving on, make sure you have Windows PowerShell 2.0 RTM on your machines.

Windows PowerShell 2.0 is included in Windows 7 and Windows Server 2008 R2 operating systems by default. (For Server Core edition, it’s available as a feature.) For down-level versions of Windows, you can download Windows PowerShell 2.0 RTM here: http://support.microsoft.com/kb/968929.

If you have installed Windows PowerShell and want to find out whether it is the latest version, or if you would like to examine your enterprise to find out which machines are running a particular version of Windows PowerShell, see: http://powershell.com/cs/blogs/tobias/archive/2010/01/21/are-you-using-the-correct-powershell-version.aspx.
Copyright

© 2010 PowerShell.com

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems except as permitted under Section 107 or 108 of the 1976 United States Copyright Act without the prior written permission of the publisher, PowerShell.com.

For permission to use material from the text please contact PowerShell.com at info@powershell.com.

Microsoft® Windows PowerShell® and Microsoft® SQL Server® are registered trademarks of Microsoft Corporation in the United Stated and other countries. All other trademarks are the property of their respective owners.
About the Authors

**Dr. Tobias Weltner** is one of the most visible Windows PowerShell MVPs in Europe. He has published more than 80 books about Windows and scripting techniques with Microsoft Press and other publishers, is a regular speaker at conferences and road shows, and provides high-level training in Windows PowerShell and scripting for companies throughout Europe.

Tobias created the powershell.com website and community in an effort to help people adopt and use Windows PowerShell more efficiently. As a software architect, he created a number of award-winning scripting tools such as SystemScripter (VBScript), the original Windows PowerShell IDE and PowerShell Plus, a comprehensive integrated Windows PowerShell development system.

For more information about Dr. Weltner's in-house training programs in Europe, email: tobias.weltner@scriptinternals.de


**Aleksandar Nikolic** has worked as a senior systems administrator for more than 16 years. He was one of the earliest adopters of Windows PowerShell, beginning with Monad, its predecessor. Now one of Europe’s most active Windows PowerShell MVPs, Aleksandar is a regular speaker at conferences in Serbia and active in user groups across Europe.

As technical editor on Dr. Tobias Welter's Windows PowerShell book, Mastering PowerShell, Aleksandar translated the book into English and served as the lead technical editor.

He is a five-time winner of the www.powershell.com activity award and a frequent contributor to the site.


**Richard Giles** has worked in computer technology for over 24 years and has a wide range of experience working in information technology including systems administration, computer consulting, and commercial software development.

Richard was part of the original development group at Mission Critical Software, a successful systems management company in Houston. After Mission Critical merged with NetIQ in 2000, Richard played a key role in the development of Operations Manager, an end-to-end service management tool that Microsoft purchased in 2000. While at NetIQ, Richard also served as the corporate architect reporting to the CTO for three years and helped define the SQL Server product line.

Richard joined Idera in 2007 as a senior developer working on SQL Server management products and is now the Senior Product Manager for Windows PowerShell products at Idera.

Acknowledgements

The authors of this book would like to extend their sincere gratitude to the following special people for making this Guide possible: Jeffrey Snover for his assistance and incredible feedback. As always, Jeffrey's insight proved invaluable as we labored to make this Guide both comprehensive and easy to use. June Blender Rogers for her brilliant technical editing skills. Shay Levy for his technical review and continued support. The authors would also like to thank Rick Pleczko and Rod Endo at Idera for their continued support of the Windows PowerShell community, David Fargo for his part in building PowerShell.com into a fantastic resource for the Windows PowerShell community, David Twamley for doing the difficult work behind the scenes and Conley Smith for her technical writing skills.
Windows PowerShell Remoting Technologies

Windows PowerShell supports two fundamentally different technologies to access remote systems. You can either use «classic» remoting which is built into individual cmdlets, or you can use Windows PowerShell Remoting to run any command on remote systems.

We will walk you through both remoting technologies to give you an overview of how they work and what they can do. Please do not be frustrated if some of the initial examples cause errors on your system. Accessing remote systems involves many security settings that might not yet be correctly configured on your system. You will learn more about fine-tuning these settings in a later part of this paper. The initial objective is to focus on the basic technology.

Classic Remote Access

With classic remote access, Windows PowerShell is not directly involved in accessing remote systems. Instead, single cmdlets implement the technology required to access remote systems independently. With this type of remoting, it is up to the cmdlet author to choose and implement the appropriate technology. In most cases, this type of remote access uses the same remoting technology present in other remote management products and tools: Remote Procedure Call (RPC) and Distributed COM (DCOM).

You can identify cmdlets with built-in remoting support by searching for cmdlets that support the `ComputerName` parameter:

```powershell
PS> Get-Help * -Parameter ComputerName
```

Some of the cmdlets this command returns are involved with Windows PowerShell Remoting. So, to identify only those that use classic remoting techniques, exclude cmdlets with a `Session` parameter. Here is an example taken from the Windows PowerShell help files:

```powershell
```

The next command uses `Get-WmiObject` to access BIOS information on `PC01`:

```powershell
PS> Get-WmiObject Win32_BIOS -ComputerName PC01
```

By default, you will be authenticated as the current user. One requirement for most remote accesses is that your user account has local administrator privileges on the target system. If you are currently logged on as a domain administrator, by default, you are also a member of the local `Administrators` group unless your security settings have been changed.

If Windows PowerShell returns an `Access Denied` error message, you can check whether the cmdlet supports the `Credential` parameter. If so, use it to log on to the target system as a different user:

```powershell
PS> Get-WmiObject Win32_BIOS -ComputerName PC01 -Credential (Get-Credential)
```

To find all cmdlets that support both the `ComputerName` and the `Credential` parameters in one parameter set, use this command:

```powershell
PS> Get-Command -CommandType cmdlet | ? { @($_.ParameterSets | ? { $p = $_.Parameters | Select-Object -Expand Name; (($p -contains 'computername') -and ($p -contains 'credential'))}).Count -gt 0}
```

<table>
<thead>
<tr>
<th>Cmdlet</th>
<th>Description</th>
<th>Supports Credential</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Clear-EventLog</code></td>
<td>clears all event log entries</td>
<td></td>
</tr>
<tr>
<td><code>Get-Counter</code></td>
<td>gets performance counter data</td>
<td></td>
</tr>
<tr>
<td><code>Get-EventLog</code></td>
<td>gets event logs and event log entries</td>
<td></td>
</tr>
<tr>
<td><code>Get-HotFix</code></td>
<td>gets hotfixes and updates</td>
<td>x</td>
</tr>
<tr>
<td><code>Get-Process</code></td>
<td>gets running processes</td>
<td></td>
</tr>
<tr>
<td><code>Get-Service</code></td>
<td>gets services</td>
<td></td>
</tr>
<tr>
<td>Cmdlet</td>
<td>Description</td>
<td>Supports Credential</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Get-WinEvent</td>
<td>gets all event log entries including the new XML-based event logs introduced in Windows Vista</td>
<td>x</td>
</tr>
<tr>
<td>Get-WmiObject</td>
<td>gets instances of WMI classes</td>
<td>x</td>
</tr>
<tr>
<td>Invoke-WmiMethod</td>
<td>calls WMI methods</td>
<td>x</td>
</tr>
<tr>
<td>Limit-EventLog</td>
<td>limits the size of event logs</td>
<td></td>
</tr>
<tr>
<td>New-EventLog</td>
<td>creates a new event log and/or an event source</td>
<td></td>
</tr>
<tr>
<td>Register-WmiEvent</td>
<td>subscribes to WMI events</td>
<td>x</td>
</tr>
<tr>
<td>Remove-EventLog</td>
<td>deletes an event log and/or event sources</td>
<td></td>
</tr>
<tr>
<td>Remove-WmiObject</td>
<td>deletes an instance of a WMI class</td>
<td>x</td>
</tr>
<tr>
<td>Restart-Computer</td>
<td>restarts the operating system</td>
<td>x</td>
</tr>
<tr>
<td>Set-Service</td>
<td>changes the settings of a service</td>
<td></td>
</tr>
<tr>
<td>Set-WmiInstance</td>
<td>creates or updates an instance of a WMI class</td>
<td>x</td>
</tr>
<tr>
<td>Show-EventLog</td>
<td>opens the event log in the Event Viewer window</td>
<td></td>
</tr>
<tr>
<td>Stop-Computer</td>
<td>shuts down a computer</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 1: Cmdlets supporting remote access technologies on their own (ComputerName parameter)

Transparent Authentication

If you do not use the Credential parameter to supply explicit credentials Windows PowerShell tries to authenticate you as the current user. This works well in a domain environment when you are domain administrator. However, in peer-to-peer networks, there is no global domain administrator.

So, if you are not a member of a domain, you can create a "pseudo domain administrator" by adding a local account to your machines which all use the same name and password. Be sure this account is member of the Administrators group on the local computer.

Then, when you log on as this user and access a remote system, you will be authenticated as this user on the remote system. Since there is a matching account on the target system, it will authenticate your local account against the matching local account on the target system, and you do not need to specify the credentials.

If Windows PowerShell still returns an Access Denied error with this technique, the machines involved might not be peer-to-peer machines after all. Machines that are joined to a domain do not support this type of transparent credentials. In this case, you either need to log on with the credentials of a domain administrator or provide the credentials explicitly.

Pitfalls and Gotchas

Classic remote techniques can be awesome when all works right. However, if you do not meet the technical requirements, it is very hard to diagnose the problem. Here is an example. Let's assume you want to get event log entries from a remote system, so you try this:

```
PS> Get-EventLog Application -EntryType Error -ComputerName storage1
```

Index | Time      | EntryType | Source    | InstanceID | Message
-----|-----------|-----------|-----------|------------|--------
51054| Jan 07 20:15| Error     | Application Error | 1001 | ...
51052| Jan 07 20:15| Error     | Application Error | 1001 | ...
(...)

6 Administrator's Guide to Windows PowerShell Remoting
All seems to work just fine. But look what happens when you disable the `RemoteRegistry` service on the target machine:

```
PS> (Get-WmiObject Win32_Service -Filter 'name="RemoteRegistry"' -Computer storage1).StopService().ReturnValue
0
```

When you now repeat the `Get-EventLog` command, you get a very confusing error message:

```
PS> Get-EventLog Application -EntryType Error -ComputerName storage1
Get-EventLog : The network path was not found.
At line:1 char:13
+ get-eventlog <<<< Application -ComputerName Storage1
```

This error message complains about a missing network path and is a good example of the type of confusing error messages you might receive when important settings or permissions are missing. This error messages won't help much in identifying the real underlying problem.

After you re-enable the `RemoteRegistry` service on the target machine, `Get-EventLog` works again.

```
PS> (Get-WmiObject Win32_Service -Filter 'name="RemoteRegistry"' -Computer storage1).StartService().ReturnValue
0
```

As you can see, error messages with classic remoting can be misleading because the errors are raised internally with the remoting technique the cmdlet chose to use internally.

Windows PowerShell Remoting

Since traditional remote access is implemented by individual cmdlets, it is inconsistent (uses different techniques and demands different requirements) and available only in selected cmdlets. The technology used for remote access can vary from cmdlet to cmdlet and is not readily known to you. Each cmdlet uses whatever remoting technology its author chose. Most cmdlets use `Remote Procedure Call` (RPC), but might also require additional services and settings on the target system.

Beginning in Windows PowerShell 2.0, there is an alternate and more universal way of accessing remote systems: Windows PowerShell Remoting. With this type of remoting, Windows PowerShell handles remote access for all commands. It transfers your commands to the remote system using the relatively new and highly configurable WinRM service, executes the code in a separate session that runs on the remote system, and returns the results to the calling system.

<table>
<thead>
<tr>
<th>Cmdlet</th>
<th>Description</th>
<th>Credential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter-PSSession</td>
<td>connects to an interactive session</td>
<td>x</td>
</tr>
<tr>
<td>Get-PSSession</td>
<td>lists all sessions</td>
<td></td>
</tr>
<tr>
<td>Invoke-Command</td>
<td>executes code in a session</td>
<td>x</td>
</tr>
<tr>
<td>New-PSSession</td>
<td>creates a new session</td>
<td>x</td>
</tr>
<tr>
<td>Remove-PSSession</td>
<td>closes and discards a session</td>
<td></td>
</tr>
<tr>
<td>Test-WSMAN</td>
<td>connects to wsman</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2: Cmdlets used to manage Windows PowerShell Remoting

With this architecture, your commands always runs locally, just on different machines, so there is no need for individual cmdlets to support remoting. Any command that can be run locally can also be run remotely. However, to make that happen, a number of prerequisites have to be in place.
Minimal Setup

If your machine is acting as client only (accessing other systems but not being accessible for other computers) in a domain environment, you do not need to configure anything special.

Note that, with this type of minimum setup, your computer cannot provide remoting services. It can only access other systems that have enabled remoting. You will not be able to test remoting on your own computer.

Also, in a peer-to-peer environment (without a domain), minimal setup will not work because you need to change a WinRM setting called TrustedHosts which is only configured when you run the full setup. So if you are running in a peer-to-peer environment or need to access non-trusted domains, run the full setup.

TIP: After you run the full setup and configure TrustedHosts, you can then undo some of the changes made by the full setup -- remove the firewall exception and stop the WinRM service -- if you only want to access other systems. This will prevent other systems from connecting to you.

If you want to set up TrustedHosts without calling Enable-PSRemoting, you need to add a registry key and then temporarily run the WinRM service. Run the following lines with full Administrator privileges:

```
PS> Set-ItemProperty HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
LocalAccountTokenFilterPolicy 1 -Type DWord
PS> Start-Service WinRM
```

If your client is running Windows XP, you also need to set another registry entry:

```
PS> Set-ItemProperty HKLM:\System\CurrentControlSet\Control\Lsa ForceGuest 0
```

Now you can add the TrustedHosts entry:

```
PS> Set-Item WSMan:\localhost\Client\TrustedHosts -Value * -Force
```

Finally, you can turn off the WinRM service again and revert the value of the LocalAccountTokenFilterPolicy registry entry:

```
PS> Stop-Service WinRM
PS> Set-ItemProperty HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
LocalAccountTokenFilterPolicy 0 -Type DWord
```

Your client is now able to access other systems remotely using Windows PowerShell Remoting and is no longer limited to Kerberos and machines inside a domain.

Full Setup

To turn your computer in a remoting server that is accessible from other machines, the prerequisites for remoting need to be in place:

- **WinRM Service**: This service receives requests from other computers and needs to be running.
- **Listener**: Inside WinRM, a listener needs to be set up that listens on the network port Windows PowerShell Remoting uses to communicate.
- **Firewall Exception**: A firewall exception is needed that allows outside requests to reach the WinRM service.

To set up these prerequisites, launch a Windows PowerShell console with elevated privileges and run this command:

```
PS> Enable-PSRemoting
```

TIP: This cmdlet fails when one of your network connections is set to Public. So, if you receive an error that complains about not being able to create the firewall exception, check the troubleshooting section later in this paper to find out how to resolve this issue.
By default, Windows PowerShell Remoting uses Kerberos authentication, which requires a domain controller. If you are not using a domain or need to connect to machines outside your own trusted domain(s), one more step is necessary:

```
PS> Set-Item wsman:\localhost\client\trustedhosts * -Force
```

This command configures Windows PowerShell Remoting so that you can connect to any computer, not just those inside your own trusted domain(s).

**TIP:** You can further harden this setting by using explicit computer names, IP addresses or prefixes instead of `*`:

```
PS> Set-Item wsman:\localhost\client\trustedhosts server_* -Force
PS> Set-Item wsman:\localhost\client\trustedhosts 10.10.10.* -Force -Concatenate
PS> Set-Item wsman:\localhost\client\trustedhosts 192.168.2.110 -Force -Concatenate
PS> Dir wsman:\localhost\client
```

This example configures `TrustedHosts` in a way that allows you access to computers with names that start with `server_*` or with IP addresses that start with `10.10.10.*` or with the particular system `192.168.2.110`.

Also, you need to add the `Concatenate` parameter to add new entries without overwriting existing ones.

These settings do not determine which computers can connect to you. `TrustedHosts` is just a reminder to you to keep you from connecting to evil machines. If you do not want this type of limitation, set `TrustedHosts` to `*`.

Why limit yourself to connecting only to machines listed in `TrustedHosts`? As long as you are using Kerberos authentication within a domain, there is no need for `TrustedHosts`, because Kerberos provides mutual authentication and assures that your credentials are never passed to a machine outside your trusted domain(s).

However, with other types of authentication, there is no guarantee the computer you connect to is not an evil machine that someone smuggled into your network. If you provide your credentials to such a machine, it could grab them and do bad things with them.

This is why, by default, Windows PowerShell Remoting does not allow connections to these machines and leaves it to you to reduce security by adding exceptions to `TrustedHosts`.

In reality, though, as a good administrator you know pretty well which machines you connect to and whether they are your own or not. So here, `TrustedHosts` should be set to `*` so you can decide which computers you want to access.

`TrustedHosts` essentially works almost like a `Force` switch. Either the system you want to connect to is already part of `TrustedHosts`, so you can connect right away, or it is not, in which case any Administrator can simply add it to `TrustedHosts`.

**Setup Recommendations**

When you use Windows PowerShell Remoting for the first time, we suggest you set up a test scenario and fully enable Windows PowerShell Remoting on all machines with these two simple commands:

```
PS> Enable-PSRemoting -Force
PS> Set-Item wsman:\localhost\client\trustedhosts * -Force
```

This way, you can focus entirely on playing with the remoting cmdlets and not having to worry about remoting setup questions. Once you have played with Windows PowerShell Remoting and feel more confident, before moving to your production systems, evaluate in more detail which settings are really required for particular machines and follow the «least privileged» security rule.
Running Remote Commands

With *Invoke-Command*, you run commands locally on a remote machine. At a minimum, this cmdlet requires the commands you want to execute. If you do not provide a computer name, it runs your commands locally on your computer, so to work remotely, you also need to specify the name of the computer you want to access.

Next, *Invoke-Command* sends a request via WinRM on port 5985 to the remote system and instantiates a new Windows PowerShell session on the remote system. After it establishes the session, it transfers the commands you provided to that session and instructs it to run the commands. Finally, the results returned by that command are transferred back to your system.

The next command executes *Get-Service* on *PC01* and uses Windows PowerShell Remoting:

```
PS> Invoke-Command { Get-Service } -ComputerName PC01
```

*Get-Service* is a cmdlet that has built-in remoting support based on classic remoting techniques, so you could have used either one remote access technologies to receive (almost) the same result:

```
PS> Get-Service -ComputerName PC01
```

However, the universal Windows PowerShell Remoting approach ensures that *any* code you want to run can run remotely by using the same procedure. For example, although *Get-Service* has a *ComputerName* parameter and would have been able to get information on its own from a remote system, it does not support the *Credential* parameter. So, to authenticate as a different user, your only choice is Windows PowerShell Remoting:

```
PS> Invoke-Command { Get-Service } -ComputerName PC01 -Credential (Get-Credential)
```

In addition, Windows PowerShell Remoting takes care of all remoting requirements and does not require your particular command or cmdlet to support remoting. You can remote any command including native applications such as *netstat* or *ipconfig*:

```
PS> Invoke-Command { netstat } -ComputerName PC01
```

**NOTE:** Windows PowerShell Remoting sends your code to the remote system and executes it inside a separate session on that system. The code runs locally, just on a different machine, so, you can remote any command that can run locally. Just make sure the command you want to run really does exist on the remote system. For example, you might have *robocopy.exe* on your local machine, but you can run it remotely on another system only if it is present on the remote system, too.

Also, all file paths, variables and locations in your commands are interpreted on the remote system, not your own. So to access folders, they must exist on the remote system.

Finally, beware of second hop issues that occur when you try to access protected resources, such as file shares. Because you have already authenticated on the remote system, your commands cannot pass your credentials to resources outside the target system. This is why your code cannot "hop" to additional machines or access file shares that require authentication. The following example illustrates an illegal second hop call:

```
PS> Invoke-Command { Get-WmiObject Win32_BIOS -ComputerName PC04 } -ComputerName PC01
```

In this case, your commands would be transferred to *PC01* and executed there. The command would then try to get WMI information from *PC04*, which requires another authentication and would fail.

To access different systems from inside a remote system, you must explicitly present new credentials or use advanced techniques such as *Delegation* or *CredSSP* to allow the target system to pass your credentials forward.
Interactive Remoting

Instead of sending code to a remote session, you can also visit a remote session and work interactively. Use `Enter-PSSession` to connect to a remote session. Your prompt changes and shows the target name in square brackets. Although you type commands in your local console, anything you enter is executed on the remote system until you leave the remote session by using `Exit-PSSession`.

```
PS> Enter-PSSession -ComputerName PC01
```

When you connect your Windows PowerShell console to a remote session, notice that your prompt might change as well as other settings you may have configured in your local session. This is because the remote session runs on another system and no longer executes any of your local profile scripts. Any adjustments and changes that you made in your local profile scripts will be missing from the remote session. The remote session is a completely separate Windows PowerShell environment.

Disabling Remoting

Enabling Windows PowerShell Remoting is a one-time setting that persists until you disable it. To disable Windows PowerShell Remoting, use this command which requires Administrator privileges:

```
PS> Disable-PSRemoting
```

**NOTE:** Disable-PSRemoting does not reverse all changes made by Enable-PSRemoting. For example, Enable-PSRemoting starts the WinRM service if it wasn't running already. Disable-PSRemoting cannot undo this change because it does not know if the WinRM service was enabled by other services in the meantime. This is why the command lists all the changes made by Enable-PSRemoting and asks you to undo part of these changes manually, if necessary.

Removing the Listener

The Disable-PSRemoting function does not remove the WSMan listener. Your computer still listens to requests coming from other machines. To remove all listeners, do this (you need full Administrator privileges to access the "localhost" folder):

```
Remove-Item WSMan:\localhost\Listener\* -Recurse -WhatIf
```

In the command, remove the `WhatIf` parameter to remove all listeners.

By removing the listener, you actually prevent the WinRM service from processing requests sent by other machines. This might be important if you continue to run WinRM for other purposes.

**IMPORTANT:** The previous command removes all listeners including listeners that were added by other applications. To remove only listeners that listen on the Windows PowerShell Remoting default port, 5985, use this command:

```
PS> Dir WSMan:localhost\Listener -Recurse | Foreach-Object { $_.PSPath } | Where-Object { (Get-Item "$_\Port").Value -eq 5985 } | Remove-Item -WhatIf
```

Again, to actually remove listeners, remove the `WhatIf` parameter from the previous command.

Disabling the WinRM Service

The WinRM service accepts remote requests. When you enable Windows PowerShell Remoting, this service is started and is set to start automatically. Disabling Windows PowerShell Remoting leaves the service running because it cannot determine whether other applications may require this service.

Generally, Windows servers, such as Windows Server 2003 or 2008 run the WinRM service by default, so, on servers, unless there is a specific reason, you should leave the WinRM service running.
By contrast, on clients, the default for the WinRM service is a manual start. So, if you enabled Windows PowerShell Remoting on a client and want to return to the initial defaults, stop the service and set it back to the manual start mode like this:

```
PS> Set-Service winrm -StartupType Manual
PS> Stop-Service winrm
```

**IMPORTANT:** WinRM might be needed by other applications. Before disabling WinRM, make sure no other application or service depends on it.

### Removing Firewall Exception

*Enable-PSRemoting* enables predefined firewall rules that allow HTTP on port 5985. When you disable Windows PowerShell Remoting, these exceptions remain. To remove them, either manually open the firewall configuration and disable the rules, or use Windows PowerShell.

Here are some functions you can use to manage the Windows firewall. *Get-FirewallRule* uses a COM object to access the rules defined in your Windows Firewall. The COM object that this function uses has been changed over time, so the function tries to use the modern approach that was introduced in Windows Vista. If that fails, it falls back to the old way that was used in Windows XP.

```powershell
function Get-FirewallRule($rule = '*', $port = '*') {
    try {
        (New-Object -ComObject HNetCfg.FwPolicy2).Rules |
        Where-Object { $_.Name -like $rule } |
        Where-Object { $_.LocalPorts -like $port }
    }
    catch [system.Management.Automation.PSArgumentException] {
        # this handles firewall rules on Windows XP:
        0..2 | Foreach-Object {
            $obj = New-Object -ComObject HnetCfg.FwMgr
        }
        try {
            $obj.LocalPolicy.GetProfileByType($_).GloballyOpenPorts |
            Where-Object { $_.Name -like $rule } |
            Where-Object { $_.Port.toString() -like $port }
        }
    }
}
```

```powershell
function Disable-FirewallRule([Parameter(Position=0, Mandatory=$true, ValueFromPipeline=$true)]$rule) {
    process {
        $rule.Enabled = $false
    }
}
```

```powershell
function Enable-FirewallRule([Parameter(Position=0, Mandatory=$true, ValueFromPipeline=$true)]$rule) {
    process {
        $rule.Enabled = $true
    }
}
```
To check for any rule on port \texttt{5985}, use \texttt{Get-FirewallRule}:

\begin{verbatim}
Get-FirewallRule -Port 5985 | Format-Table Name, Enabled, Profiles, LocalPorts, Description -Auto
\end{verbatim}

To disable these rules, pipe the rules to \texttt{Disable-FirewallRule}:

\begin{verbatim}
Get-FirewallRule -Port 5985 | Disable-FirewallRule
\end{verbatim}

\textbf{IMPORTANT}: These functions will disable \textit{any} rule on port 5985. By default, WinRM is the only service that uses this port. There could be other services using this port if they were added manually. Thus, you may want to filter the results to be even more specific. To select only rules that use port 5985 \textit{and} the HTTP protocol, try this:

\begin{verbatim}
Get-FirewallRule -Port 5985 | Where-Object { $_.Protocol -eq 6 }
\end{verbatim}

There are additional firewall rules that can be enabled manually to extend the reach of Windows PowerShell Remoting. For example, there are compatibility rules that allow Windows PowerShell requests on port 80, and there are additional rules for HTTPS on port 5986. Neither of these rules is enabled by default when you run \texttt{Enable-PSRemoting}.

Play with \texttt{Get-FirewallRule} to get a better picture what might be enabled on your system:

\begin{verbatim}
PS> Get-FirewallRule -Port 80
\end{verbatim}

Unfortunately, firewall rules are localized so their names might differ depending on the locale that your system uses. Use the next command to list all rules associated with Windows PowerShell Remoting:

\begin{verbatim}
PS> Get-FirewallRule *Windows?Remote*
\end{verbatim}

---

**Understanding Windows PowerShell Sessions**

Windows PowerShell sessions are the core building blocks of Windows PowerShell Remoting so it is a good idea to get a better understanding what they are and what they do.

Whenever you launch a new Windows PowerShell console, you are actually creating a Windows PowerShell session. A Windows PowerShell session essentially is a place where Windows PowerShell code can live. Without remoting, all sessions run locally on your own machine. They are independent environments and cannot interact.

With Windows PowerShell Remoting, you gain a way to create such sessions anywhere, locally as well as on remote systems, and keep a connection between them. So Windows PowerShell Remoting is, to some extent, a marshaling service that enables you to send code to another session and get results.

**Temporary Sessions**

Whenever you use Windows PowerShell Remoting with the \texttt{ComputerName} parameter, a temporary session is created for you and used only for this particular command. Once the command completes, the session is automatically closed and discarded.

In the following example, each \texttt{Invoke-Command} command creates a separate and independent session, which is why the variable defined in the first command is no longer available to the second command:

\begin{verbatim}
PS> Invoke-Computer { $env:computername; $test=1 } -ComputerName PC01
PS> Invoke-Computer { $env:computername; $test } -ComputerName PC01
\end{verbatim}

Temporary sessions are a convenient way for quick remote access, but whenever you plan to access a remote system more than once, they are very inefficient and slow because Windows PowerShell needs to create and discard sessions for every single command.
Reusable Sessions

A more efficient way is to create sessions yourself and keep them around until you no longer need them. Then, whenever you need remote access, you present the existing session and use it, instead of creating a new one. The next command creates a session to PC01 and stores it in a variable called $session:

PS> $session = New-PSSession -ComputerName PC01

To use this session for a particular remoting command, use the Session parameter instead of ComputerName:

PS> Invoke-Computer { $env:computername; $test=1 } -Session $session
PS> Invoke-Computer { $env:computername; $test } -Session $session

Here, both calls share the same session, which is why the variable defined in the first command is still present in the second command. Also, these commands run almost instantaneously because there is less overhead; no session needs to be created or discarded. In essence, reusable sessions work similar to a remote desktop connection that you just disconnect and reconnect without actually logging off or on.

To check the status of a session, dump it. It tells you whether the connection is still alive and available, and which computer the session runs on:

PS> $session

You can also use Get-PSSession to dump all sessions you generated or pick a specific session from the pool of available sessions.

It is your responsibility to close and discard sessions that you no longer need. Use Remove-PSSession:

PS> Remove-PSSession $session

You can also remove all sessions with one command:

PS> Remove-PSSession *

All sessions that you create live in your own local Windows PowerShell session, so when you close Windows PowerShell, all remaining sessions are discarded automatically.

Session Best Practice

Often, you only need to access remote systems once. In this case, it is best to use temporary sessions because these sessions are automatically closed and removed after the call.

If you start to access remote systems more often or would like to keep the session state, use reusable sessions. The caveat is to make sure you clean up behind yourself and close these sessions when they are no longer used. Otherwise, you might quickly run out of sessions, because there is a limit of 5 concurrent sessions that any one user can open to any one remote system.

Accessing Multiple Computers: Fan-Out

While there is a limit of 5 sessions that you can simultaneously maintain on any one remote computer, there is no limit -- other than resources -- to how many sessions you can simultaneously maintain in general.

You can create sessions on many computers at the same time and then use all of these sessions together to execute code simultaneously on many computers. This is what is known as »Fan-Out«. It can help you manage server farms and configure or analyze groups of computers with just one command.

Again, you have the choice of using temporary sessions or use persistent sessions. To execute commands on many machines using temporary sessions, use the ComputerName parameter and supply a comma-separated list of computer names or IP addresses. Windows PowerShell Remoting creates a separate session for each computer, and Invoke-Command runs the code in all sessions at the same time.

PS> Invoke-Command { $env:computername } -ComputerName demo05, localhost, storage1

You will get error messages for any computer Windows PowerShell cannot access, and the results returned by the commands on all other systems.
Alternately, you can create the sessions by using `New-PSSession` and then by submitting the sessions to `Invoke-Command`. The next command creates sessions on all computers listed. You will get an error message for any computer that Windows PowerShell cannot access.

```
PS> $session = New-PSSession storage1, localhost, demo5
```

All successfully created sessions are saved in `$session`:

```
PS> $session
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>ComputerName</th>
<th>State</th>
<th>ConfigurationName</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Session1</td>
<td>localhost</td>
<td>Opened</td>
<td>Microsoft.PowerShell</td>
<td>Available</td>
</tr>
<tr>
<td>2</td>
<td>Session2</td>
<td>demo5</td>
<td>Opened</td>
<td>Microsoft.PowerShell</td>
<td>Available</td>
</tr>
<tr>
<td>3</td>
<td>Session3</td>
<td>storage1</td>
<td>Opened</td>
<td>Microsoft.PowerShell</td>
<td>Available</td>
</tr>
</tbody>
</table>

As is always the case with Windows PowerShell cmdlets, the result really is an array if the cmdlet returned more than one result. So `$session` is not a »super session« targeting multiple computers. Instead, it is just a container for many individual sessions. Each session always represents just one Windows PowerShell environment on one particular machine.

Since `Invoke-Command` accepts both individual sessions and array containing multiple sessions, you can now execute a command on all sessions simultaneously. The next command creates a registry key (`HKEY_LOCAL_MACHINE\Software\Test`) on all computers on which sessions have been established:

```
PS> Invoke-Command { Md HKLM:\Software\Test } -Session $session
```

### Throttle Limit

When you access multiple sessions at the same time, a lot of data might be sent from those sessions to your machine. To avoid being overwhelmed and to keep network bandwidth in safe bounds, there is a built-in throttle limit of 32. Only 32 connections are serviced simultaneously.

If you specify more, then Windows PowerShell parks the remaining requests in a queue and waits for the first 32 sessions to complete. After the first session returns their results, the next session in the queue is used. In essence, there are never more than 32 simultaneous commands executing.

Depending on what you want to do and your available network bandwidth, you can adjust the throttle limit with the `ThrottleLimit` parameter. Lower it on slow networks, or increase it to speed up processes in a fast network.

### Managing Sessions and Releasing Resources

Each session that you create with `New-PSSession` uses a certain amount of resources. If you open too many sessions at the same time, you might run out of resources or allowable connections and at some point will be unable to create any more new sessions.

To check your sessions, use `Get-PSSession`:

```
PS> Get-PSSession
```

To close and discard sessions you no longer need, use `Remove-PSSession`. You can select sessions to remove either by name or by ID:

```
PS> Remove-PSSession -Name Session4
PS> Remove-PSSession -Id 1
```

To close and remove all sessions in one call, use this command:

```
PS> Remove-PSSession *
```

How is session lifetime handled by Windows PowerShell Remoting if you do not clean up? Let's assume you connect to a remote system and start a couple of sessions. Next, you disconnect your machine from the network and walk home. Will your stranded sessions stay alive forever on the server?
Actually, each session is a separate process that you can see when you look in the right place:

```
PS> $s = New-PSSession -ComputerName $env:computername
PS> Get-Process wsmprovhost
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
</tr>
</thead>
<tbody>
<tr>
<td>206</td>
<td>34</td>
<td>45308</td>
<td>49864</td>
<td>562</td>
<td>0,47</td>
<td>2712</td>
<td>wsmprovhost</td>
</tr>
</tbody>
</table>

Each session that you create is represented by a process named `wsmprovhost`. Removing this process also removes the session:

```
PS> $s
```

```
Id Name            ComputerName    State    ConfigurationName     Availability
-- ----            -----------      -----    -----------------------    --------
2 Session2        demo5           Opened   Microsoft.PowerShell     Available
```

```
PS> Stop-Process -Name wsmprovhost
PS> $s
```

```
Id Name            ComputerName    State    ConfigurationName     Availability
-- ----            -----------      -----    -----------------------    --------
2 Session2        demo5           Broken   Microsoft.PowerShell     None
```

After the process is killed, the session state changes to »broken« and is no longer available.

The same occurs automatically if a remote session loses its connection to the client. Although Windows PowerShell Remoting really acts like a web service and uses stateless connections, the server maintains a heartbeat to the client. If the network connection between client and server is lost for more than approximately three minutes, the server discards the session. This is controlled by a WSMan setting called `IdleTimeout`:

```
PS> Dir WSMan:\localhost\shell\idletimeout
```

```
<table>
<thead>
<tr>
<th>IdleTimeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>180000</td>
</tr>
</tbody>
</table>
```

It returns the number of milliseconds a connection can be disconnected before the server discards the session.

```
PS> "Idle Timeout = @{$((Dir WSMan:\localhost\shell\idletimeout).Value/60000) minutes}"
```

```
Idle Timeout = 3 minutes
```

**Implicit Remoting**

All of the previous examples used explicit remoting, where you explicitly instructed `Invoke-Command` to run commands in another session. Another way of remoting is `implicit remoting` in which the command knows that it should run in another session.

Implicit remoting is really just an extra layer of convenience. It uses the same technology and is based on sessions.

The first thing you need for implicit remoting is a session.

```
PS> $session = New-PSSession -ComputerName WIN-KHHRLOLOTMS3 -Credential (Get-Credential)
```

After you have a session, you can load additional snap-ins or modules or create your own functions in the session.

In the next example, you enter a session that you created. Next, you look at the available modules and import one. In addition, you define a new simple function.
NOTE: This is a real-world scenario. In this example, I am accessing a Windows Server 2008 R2 which is the first server operating system that ships with Windows PowerShell 2.0 built-in. In addition, this server comes with Windows PowerShell-based management tools allowing you to add or remove server features.

These management commands are available only locally on that server. With Windows PowerShell Remoting and implicit remoting, you can also use these cmdlets on your own computer. This illustrates how flexible server management can be. Instead of having to install management tools on your own computer, Windows PowerShell Remoting brings the server cmdlets to you.

# Create a session
PS> Enter-PSSession $session
[win-kkhrlotms3]: PS C:\Users\Tobias\Documents> Get-Module -ListAvailable

ModuleType Name                      ExportedCommands
---------- ---- -------------------
Manifest  ActiveDirectory           {}
Manifest  ADRMS                     {}
Manifest  AppLocker                 {}
Manifest  BestPractices             {}
Manifest  BitsTransfer              {}
Manifest  PSDiagnostics             {}
Manifest  ServerManager             {}
Manifest  TroubleshootingPack       {}

# Import a module and examine the commands in the module
[win-kkhrlotms3]: PS C:\Users\Tobias\Documents> Import-Module ServerManager
[win-kkhrlotms3]: PS C:\Users\Tobias\Documents> Get-Command -Module ServerManager

CommandType     Name                                 Definition
------------     -----                                 --------
Cmdlet          Add-WindowsFeature                   Add-WindowsFeature [-Name] <Feature[]> [-Include...
Cmdlet          Get-WindowsFeature                   Get-WindowsFeature [[-Name] <String[]>] [-LogPat...
Cmdlet          Remove-WindowsFeature                Remove-WindowsFeature [-Name] <Feature[]> [-LogP...

# Create a function and exit the session
[win-kkhrlotms3]: PS C:\Users\Tobias\Documents> Function test { "I am executing on $env:computername." }
[win-kkhrlotms3]: PS C:\Users\Tobias\Documents> Exit-PSSession
PS>

When you leave the remote session and return to your own local session, the remote session is still available in $session. You have imported the ServerManager cmdlets and defined a function called test in the session.

You can import the Test function from the remote session. To do that, use Import-PSSession and specify the command you want to import:

PS> Import-PSSession $session -CommandName test

IMPORTANT: The command you selected is imported into your local session as a script. Importing commands from a session only works if you have enabled script execution on your machine. If you get an error message instead, use Get-ExecutionPolicy and make sure the execution policy is set to RemoteSigned.
Now, the `Test` function is available in your local session. When you call it, you get the result. As you can see from the result, the script block that Test invokes runs on the server, not on your system. This is an example of implicit remoting, because the local command really runs on a different machine without having to explicitly specify this.

```
PS> test
I am executing on WIN-KHHRLLOTMS3.
```

In essence, implicit remoting hasn't enabled you to do something completely new. It is just a convenient way of calling remote commands. Without implicit remoting, you could have achieved the same result with `Invoke-Command`:

```
PS> Invoke-Command { test } -Session $session
I am executing on WIN-KHHRLLOTMS3.
```

Implicit remoting is a great way of making server-side cmdlets available on client computers. For example, to manage the Server 2008 R2 that I used in this example, I can import all cmdlets from the `ServerManager` module that I imported into my remote session:

```
PS> $result = Import-PSSession $session -Module ServerManager
I captured the result of `Import-PSSession` in a variable so I have the list of imported cmdlets when I need it:
PS> $result ExportedCommands

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove-WindowsFeature</td>
<td>Remove-WindowsFeature</td>
</tr>
<tr>
<td>Get-WindowsFeature</td>
<td>Get-WindowsFeature</td>
</tr>
<tr>
<td>Add-WindowsFeature</td>
<td>Add-WindowsFeature</td>
</tr>
</tbody>
</table>

Thanks to implicit remoting, I can use the `ServerManager` cmdlets on my local machine. Behind the scenes (implicitly), they are executed on the server. Whenever I need to add or remove features on that server, I can use `Get-WindowsFeature` to check certain features, such as the XPS-Viewer:

```
PS> Get-WindowsFeature xps*

RunspaceId       : bc04f492-3d30-4dee-b8b6-7d91da4eb72b
DisplayName      : XPS-Viewer
Name             : XPS-Viewer
Installed        : False
FeatureType      : Feature
Path             : XPS-Viewer
Depth            : 1
DependsOn        : {}
Parent           : 
SubFeatures      : {}
SystemService     : {}
Notification     : {}
BestPracticesModelId :
AdditionalInfo   : {NumericId}

The value of False for the `Installed` property indicates that this particular feature is not yet installed. Note that Windows PowerShell Remoting executed the command on the server and returned the results to my local session. Everything happens behind the scenes.
To add the XPS-Viewer, I can use Add-WindowsFeature. Because I want to determine whether installing this feature has unwanted side effects, I first use the WhatIf parameter to simulate the installation:

```powershell
PS> Add-WindowsFeature xps-viewer -WhatIf

(...)
RunspaceId    : 3e019a31-a1f8-40d1-9562-38a97a5ac622
Success       : True
RestartNeeded : Maybe
FeatureResult : {}
ExitCode      : Success
```

In RestartNeeded, I can now discover whether this particular installation requires a server restart. Well, sometimes. If the cmdlet cannot tell for sure, it returns »Maybe« which sounds nicer than »Unknown« but means essentially the same.

To actually install the feature, I run the command again, but remove -WhatIf. This time, you see a progress bar that illustrates the installation progress and, when the command completes, it returns the status. The RestartNeeded property tells you for sure whether a restart is needed and the Success property reports that the installation went smoothly.

```powershell
PS> Add-WindowsFeature xps-viewer

RunspaceId    : 3e019a31-a1f8-40d1-9562-38a97a5ac622
Success       : True
RestartNeeded : No
FeatureResult : {XPS-Viewer}
ExitCode      : Success
```

### Implicit Remoting Gotchas

Importing commands from a session imposes some gotchas that you should know about. If you try to import a command that also exists in your local session, the import fails unless you add the AllowClobber parameter, in which case the local command is no longer available. Or, you could use the Prefix parameter and specify a prefix. The imported commands are renamed. The prefix that you specify is added to the noun part of the command to differentiate it from local commands.

If you specify a command that cannot be found in the session, you get an error message complaining that Get-Command cannot find the command.

Implicit remoting relies on the session you from which you imported the commands. If the session is not available, because, for example, you removed it or the target machine rebooted, the imported command(s) no longer work.

Because imported commands are imported by executing scripts, you can only import commands when script execution is permitted. By default, the execution policy on the computer does not permit script execution. You need to adjust your execution policy first.

### Understanding Session Configurations

When you create a new session, behind the scenes, a so-called session configuration is used to initialize the session. Windows PowerShell creates at least one default session configuration when you enable Windows PowerShell Remoting. You can use Get-PSSessionConfiguration to view the existing session configurations, but you must start Windows PowerShell with the “Run as Administrator” option.
The default session configuration is called `microsoft.powershell`. On 64-bit machines, there will also be a second session configuration called `microsoft.powershell32` that runs on the 32-bit Windows PowerShell host.

### NOTE:
Default session configurations are created and secured by *Enable-PSRemoting*. If you enable Windows PowerShell Remoting manually or by using `Set-WSManQuickConfig`, there are no default session configurations.

You do not need to fiddle with session configurations if you do not feel like it. In fact, you can ignore them altogether if all you want to do is work remotely. However, taking a closer look at session configurations can be important if you want more control over the sessions that you (and others) create.

For example, you can use session configurations to control how remote commands are executed. Here are some things session configurations can do for you:

- **Pick Host:** If you have a script that uses snap-ins or COM objects that run only on 32-bit systems, and you need to run this script on a 64-bit machine, you can force Windows PowerShell to use the 32-bit host by specifying the predefined `microsoft.powershell32` session configuration that is created by default on 64-bit machines.

- **Restrict Session:** If you want to harden security, you can create a new session configuration that removes unwanted Windows PowerShell commands and, thereby, creates a restricted environment.

- **Custom Environments:** Similarly, if you want sessions to be preconfigured with snap-ins, modules or other settings, add a profile script to a new session configuration. Any time you create a session with this session configuration, the profile script runs and initializes the session, pretty similar to how your local profile scripts work.

### How Session Configurations Are Assigned

When you run a Windows PowerShell Remoting cmdlet that creates a new remote session, such as *Invoke-Command*, *Enter-PSSession* or *New-PSSession* - Windows PowerShell automatically uses the default session configuration `microsoft.powershell` unless you specify a different session configuration or have secured session configurations.

### TIP:
Whenever you run a remoting cmdlet without specifying a session configuration, Windows PowerShell uses the session configuration defined in the preference variable `$PSSessionConfigurationName`. The session configuration that is used on the server side is controlled by the client side. This makes sense; you (the caller) choose which session configuration you need.

The server side can control session configurations in a different way: by applying access security to session configurations, the server controls which session configurations are available to clients. Applying access controls on session configurations is discussed below.

If you ignore session configurations, you get a 32-bit session if the target system is a 32-bit system and a 64-bit session if the target system is a 64-bit system - much like launching Windows PowerShell interactively.

You can use the `ConfigurationName` parameter to pick a session configuration manually. For example, to create a remote session on a 64-bit system running in 32-bit, use this command:

```powershell
PS> $session32 = New-PSSession -ComputerName Win2k8R2 -ConfigurationName microsoft.powershell32
```

### Creating New Session Configurations
Session configurations can help initialize sessions with certain defaults. For example, every time you launch Windows PowerShell locally, profile scripts run. These profile scripts can change the prompt, load additional commands and configure your workspace. When you use sessions with the default session configurations, no profile scripts run.

Let's create a new session configuration that loads one of the default profile scripts. Just make sure that this file is accessible to the user who creates the session. In this example, we pick the machine level profile stored in $PsHome\Profile.ps1.

**TIP:** You can run any script in the session configuration. To change the script, simply change the script path in the command. The profile script must be present on the remote machine, so you might want to verify that it exists before using the session configuration. On the machine on which you want to register the new session configuration, launch a Windows PowerShell console with full Administrator privileges, and enter this command:

```
PS> Notepad $PsHome\Profile.ps1
```

If the profile script does not yet exist, Notepad asks whether you want to create it. In this example, we use a simple profile script that contains only two commands:

```
function prompt { "PS> " }
$host.UI.RawUI.CursorSize = 80
```

It redefines the prompt and sets the cursor size to 80%.

To register a new session configuration, run `Register-PSSessionConfiguration`. This cmdlet works locally only, that is, it creates the session configuration on the local machine, so you need to run it on the target side with full Administrator privileges:

```
PS> Register-PSSessionConfiguration -Name WithProfile -StartupScript $PsHome\Profile.ps1
```

If there is a firewall running on the local machine, the script must be present on the remote machine, so you might want to verify that it exists before using the session configuration.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Register-PSSessionConfiguration" on Target "Name: WithProfile. This will allow administrators to remotely run Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
```

```
WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin
```

```
Name                      Type                 Keys
----                      ----                 -----
WithProfile               Container             {Name=WithProfile}
```

Confirm

```
Are you sure you want to perform this action?
Performing operation ""Restart-Service"" on Target "Name: WinRM".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
```

Next, make sure the session configuration was added:

```
PS> Get-PSSessionConfiguration
```

```
Name                      PSVersion  StartupScript        Permission
----                      --------    ---------------        ---------
microsoft.powershell      2.0        BUILTIN\Administrators Access Allowed
microsoft.powershell32    2.0        BUILTIN\Administrators Access Allowed
WithProfile               2.0        C:\Windows\System...
```

21 Administrator's Guide to Windows PowerShell Remoting
Using Custom Session Configurations

After you add the new session configuration to the target computer, clients can create sessions on this target computer using the new session configuration. Simply request the session configuration by using the ConfigurationName parameter. If you have no test network, you can test this on your local machine, provided that you have enabled Windows PowerShell Remoting and run Windows PowerShell with full Administrator privileges (for example, right-click PowerShell in your start menu and choose “Run as Administrator”).

First, try to connect to an interactive remoting session without the new session configuration:

```
C:\Windows\system32 PS> enter-pssession -ComputerName $env:computername
[demo5]: PS C:\Users\w7-pc9\Documents> Exit-PSSession
C:\Windows\system32 PS>
```

You can connect to the remote session. Nothing special changes because the remote session does not run any profile script. Next, use the new WithProfile session configuration that you created earlier:

```
C:\Windows\system32 PS> Enter-PSSession -ComputerName $env:computername -ConfigurationName WithProfile
[demo5]: PS> Exit-PSSession
C:\Windows\system32 PS>
```

This time, the prompt changed, because the session was initialized with the profile script that was specified in the session configuration. Also, your Windows PowerShell cursor changes and appears as a block cursor.

**NOTE:** After you exit the remote session, the prompt reverts to its original style, but the cursor size remains as an 80% block cursor. Why?

Any changes you make by using $host.UI.RawUI, such as setting the cursor size, buffer size, or console colors, do not change Windows PowerShell settings; they change the current console settings. When the remote session ends, all functions that are defined in the session are reset, this is why you get your old prompt back. However, the console stays the same, so changes to the cursor size are not reversed. This is not related to how sessions or Windows PowerShell Remoting work. It happens because interactive remote sessions share the same console with local Windows PowerShell sessions.

Enumerating Remote Session Configurations

In the previous example, you connected to a remote session by using a custom session configuration registered on the target system. This was easy because you set up the new session configuration yourself, so you knew its name. What if you need to remotely access a system you do not know that well? How do you know about the session configurations that are available on that machine?

To list session configurations on a remote system, use Connect-WSMAN to connect to the target system and then use the WSMAN: drive to explore the session configurations available on that machine. This example finds the session configurations available on a system called storage1:

```
PS> Connect-WSMAN storagel
PS> cd WSMAN:
PSSession: stora1
PS WSMAN:\> dir

WSManConfig:

ComputerName       Type
------------------ ----
localhost          Container
storagel           Container
```

22 Administrator's Guide to Windows PowerShell Remoting
PS WSM:\> dir .\storage1\plugin

PS ManConfig: Microsoft.WSMan.Management\WSMan::Server01\Plugin

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Forwarding Plugin</td>
<td>Container</td>
<td>{Name=Event Forwarding Plugin}</td>
</tr>
<tr>
<td>microsoft.powershell</td>
<td>Container</td>
<td>{Name=microsoft.powershell}</td>
</tr>
<tr>
<td>microsoft.powershell32</td>
<td>Container</td>
<td>{Name=microsoft.powershell32}</td>
</tr>
<tr>
<td>SEL Plugin</td>
<td>Container</td>
<td>{Name=SEL Plugin}</td>
</tr>
<tr>
<td>WithProfile</td>
<td>Container</td>
<td>{Name=WithProfile}</td>
</tr>
<tr>
<td>WMI Provider</td>
<td>Container</td>
<td>{Name=WMI Provider}</td>
</tr>
</tbody>
</table>

Registered session configurations show up as plug-ins to WS-Management.

PS> Get-WsManInstance winrm/config/plugin -Enumerate -ComputerName storage1 | Select name

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Forwarding Plugin</td>
</tr>
<tr>
<td>microsoft.powershell</td>
</tr>
<tr>
<td>microsoft.powershell32</td>
</tr>
<tr>
<td>SEL Plugin</td>
</tr>
<tr>
<td>WithProfile</td>
</tr>
<tr>
<td>WMI Provider</td>
</tr>
</tbody>
</table>

Securing Session Configurations

You can assign session configurations to users automatically. For example, you may want to restrict the commands in a session that are available to some users. This is done in a two-step-process:

1. Create a new session configuration that restricts the session to only a subset of commands.
2. Change security access permissions so that the intended users can only access the new session configuration.

Creating a Restricted Session

First, create a new session configuration that restricts the session to only a safe subset of commands. To do this, run an initialization script in the session configuration that restricts the session.

Open an editor with full Administrator privileges (i.e. run notepad from an elevated Windows PowerShell console) and enter this:

```powershell
# Disable access to all applications

# Disable access to scripts
$ExecutionContext.SessionState.Scripts.Clear()

# Define a list of allowed commands

# Make everything except the allowed commands private (not visible)
Get-Command | Where-Object {$Commands -notcontains $_.Name} | ForEach-Object {$_.Visibility="Private"}
```
Save this `%windir%\system32\WindowsPowerShell\v1.0\restricted.ps1`. To make sure the script runs correctly, run it in a local Windows PowerShell console. If you did everything right, it will restrict this session and allow only the safe subset of commands defined in that script to run in the session:

```powershell
PS> & %PSHome%\restricted.ps1
```

Next, create a new session configuration that uses your script to restrict the session:

```powershell
PS> Register-PSSessionConfiguration -Name Restricted -StartupScript %PSHome%\restricted.ps1
```

### Launching a Restricted Session Manually

When the new session configuration is in place, remote users can access your computer by using the new session configuration to create a restricted session:

```powershell
PS> Enter-PSSession -ComputerName $env:COMPUTERNAME -ConfigurationName Restricted
```

By default, only members of the Administrators group on the computer have **Execute** permission to the default session configurations. You can change security descriptors on any session configuration, even on the default one.

To give a group of users permission to connect to the computer remotely only by using a specific session configuration, use `Set-PSSessionConfiguration` to add **Execute** permissions for those users to the security descriptors of the session configurations. To set the permissions, use the `SecurityDescriptorSDDL` or
ShowSecurityDescriptorUI parameter. The latter opens a familiar GUI where you can reconfigure permissions easily.

The next command opens the property page for the security descriptor of the new Restricted session configuration.

PS> Set-PSSessionConfiguration -Name Restricted -ShowSecurityDescriptorUI

**IMPORTANT:** When you create restricted sessions, be careful when choosing the cmdlets (and functions) you exclude; otherwise, the restricted session might be useless. Some cmdlets have dependencies, so, for example, Import-PSSession requires a number of other cmdlets to successfully import commands from a session. If you restrict the session too much, you will see error messages like this one:

"Import-PSSession : Import-PSSession cmdlet needs the following commands in the remote session: Get-Command, Get-FormatData, Select-Object. The following commands are used, but optional: Get-Help, Measure-Object. Please make sure the remote session includes the required commands and try again."

---

**Understanding WSMan Settings**

WSMan controls how Windows PowerShell Remoting works. By default, it uses HTTP as transport, uses port 5985, accepts only Kerberos authentication and allows a maximum of 5 remote shells per user. However, WSMan is highly configurable and can be changed to suit your needs.

All WSMan settings are managed through a virtual Windows PowerShell drive called WSM:\. You need Administrator privileges to view and change these settings, and the WinRM service needs to be running.

To illustrate how you read and change WSMan settings, here are some examples of how to change the maximum number of shells and the port used by Windows PowerShell Remoting.

**Using More Than 5 Sessions**

By default, every user can open a maximum of five sessions to any remote computer. When you try to open more, Windows PowerShell returns an error message:

```
1..10 | ForEach-Object { New-PSSession -ComputerName storage1 }
```

This command opens five sessions on the storage1 machine and then returns errors telling you that the maximum number of sessions is reached.

This restriction is a good thing, because it prevents you from carelessly wasting server resources by opening numerous sessions without closing them. When you run into an error like this, you should run Get-PSSession to check the already opened sessions, then remove any sessions you no longer need by running Remove-PSSession.

If you must use more than five simultaneous sessions, you can also change the limit. Here is how to read the currently active limit (again, to access the localhost "folder", you need full Administrator privileges):

```
Dir WSMan:\localhost\shell
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowRemoteShellAccess</td>
<td>true</td>
</tr>
<tr>
<td>IdleTimeout</td>
<td>1800000</td>
</tr>
<tr>
<td>MaxConcurrentUsers</td>
<td>5</td>
</tr>
<tr>
<td>MaxShellRunTime</td>
<td>2147483647</td>
</tr>
<tr>
<td>MaxProcessesPerShell</td>
<td>15</td>
</tr>
<tr>
<td>MaxMemoryPerShellMB</td>
<td>150</td>
</tr>
<tr>
<td>MaxShellsPerUser</td>
<td>5</td>
</tr>
</tbody>
</table>
MaxConcurrentUsers limits the number of concurrent users that can have a session on this computer. MaxShellsPerUser controls how many sessions each user can open, so, by default, a maximum of 5 users can open a maximum of 5 sessions (25 sessions).

To change a setting, use Set-Item. For example, to change the maximum number of shells per user, use this command:

```
Set-Item WSM:\localhost\shell\MaxShellsPerUser 10
```

### Using a different port

By default, Windows PowerShell Remoting uses port 5985, but you can change this default to any other port. The currently used port is part of the listener:

```
PS> dir WSM:\localhost\listener\*\Port
```

To change to a different port, use Set-Item again:

```
PS> Set-Item WSM:\localhost\listener\*\Port 1000
```

**IMPORTANT:** When you change the port assignment, you also have to add a new Firewall rule allowing other computers to access WinRM via the the port, unless the new port is already listed in other Firewall rules.

Now, all requests to this machine on port 5985 fail because the listener does not listen on port 5985 anymore. Instead, it expects requests to be directed to port 1000.

To access the machine, you would have to add the Port parameter to all of your remoting calls.

```
$pid
Invoke-Command { $pid } -ComputerName $env:computername -Port 1000
```

Your local Windows PowerShell session returns different values for $pid. The first one is the process ID of your local session. The second one is the process ID of your remote session.

### Changing WSMan: Settings Remotely

In the previous examples, you changed the settings on your local machine. However, you can also manage WSMan settings remotely. Simply use Connect-WSMan to connect to the remote machine. After a connection is established, your WSMan: drive shows the remote systems as well:

```
PS> Connect-WSMan storage1
PS> Dir WSMan:

WSManConfig:

ComputerName          Type
--------------------- ----
localhost              Container
storage1               Container
```

To change settings on the remote system, use the same technique as before, but replace paths accordingly. For example, to read the number of maximum shells on the remote system, use this command:

```
PS> Dir WSMan:\storage1\shell\MaxShellsPerUser

WSManConfig: Microsoft.WSMan.Management\WSMan:\storage1\Shell

Name          Value
-----         ------
MaxShellsPerUser  5
```
To read this setting for all computers you are connected to, use this:

```powershell
PS> $row = @{Name='Computer';Expression={$_.PSPath.Split("::")[-1].Split('\')[-0]}}
PS> Dir WSM\shell\MaxShellsPerUser | Select-Object name, $row, Value
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Computer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxShellsPerUser</td>
<td>localhost</td>
<td>10</td>
</tr>
<tr>
<td>MaxShellsPerUser</td>
<td>storage1</td>
<td>5</td>
</tr>
</tbody>
</table>

### Windows PowerShell Remoting Requirements and Gotchas

Windows PowerShell Remoting offers great flexibility yet has limitations as well:

- **Windows PowerShell 2.0 required**: To use Windows PowerShell Remoting, Windows PowerShell 2.0 needs to be present on both ends. You cannot access systems that do not have Windows PowerShell 2.0.

- **Resulting Objects are Clones**: When Windows PowerShell sends back results from another session to the caller, these objects are temporarily converted to XML. This serialization is necessary to send objects back to the caller. So, the objects you receive from Windows PowerShell Remoting are no longer the live objects. Serialization removes all methods and can change the data type of object properties.

- **Sessions are fragile**: Sessions are the heart and soul of Windows PowerShell Remoting and represent a Windows PowerShell instance. When a session is destroyed, you cannot easily recreate it like a broken network connection. For example, when a remote system reboots, all sessions are lost and all remoting calls and imported commands break.

### Windows PowerShell 2.0 required on both sides

The most obvious limitation is the way Windows PowerShell Remoting works: it sends commands from your session to another session, so the machine you are remotely accessing must have Windows PowerShell 2.0. This imposes restrictions on the type of systems you can use with Windows PowerShell Remoting. Windows PowerShell 2.0 is not available on Windows 2000, and there may be operational restrictions that prevent you from installing Windows PowerShell 2.0 on some production servers.

This limitation will have less effect over time when Windows PowerShell 2.0 becomes available on all systems in your infrastructure. With Windows 7 on the client side and Windows Server 2008 R2 on the server side, Windows PowerShell 2.0 is an integral part of modern Windows-based infrastructures.

### Resulting Objects are Clones

The objects that Windows PowerShell Remoting returns to you are not the real objects, like those that are returned by commands that run in your local session. Instead, Windows PowerShell Remoting needs to squeeze the original objects into XML to transport them back to the caller. This temporary conversion changes the objects significantly. Because Windows PowerShell Remoting occurs in a session that is completely independent of your local session, the objects that are returned lose their connection to the target system. This has three consequences:

- **No Methods**: Because the objects that Windows PowerShell Remoting returns have no connection to the system on which they were created, they cannot execute commands to change the target system. This is why the object clones do not have methods.
- **Read-Only Properties:** Because the objects that Windows PowerShell Remoting returns have no connection to the system on which they were created, changes to object properties have no effect on the target system. In essence, all object properties are read-only.

- **Changing Property Types:** When objects are converted to XML for transport, all object content is serialized in XML-compatible way. Most data types can be preserved, and when the serialized XML is deserialized (turned back into objects), these properties keep their original data type. However, if an object property stores data in special types that Windows PowerShell does not recognize and does not know how to convert back and forth, the property content is converted to plain text. Because of this, some property data types might change. For example, a mailbox size that originally was in some numeric format might now appear as formatted text.

All of these limitations occur only after the objects are returned from the remote session to your local session. As long as the objects stay in the remote session, there is no need for XML transformation and the resulting object changes. So, the best practice is to keep the results in the remote session for as long as possible. The more data that can be processed in the remote session, the better. Having to return data from a remote session to your calling session not only cripples the objects to some extent, it is also «expensive» in time and resources. The less data you transfer, the better.

### How Objects Travel Back

In order for Windows PowerShell to transfer results from a remote session back to your own local session, the objects need to be temporarily converted to CliXML (Constrained Language in XML). CliXML is pure text and can easily be transferred across any transport. On the caller side, the received CliXML is then converted back into objects.

This conversion procedure is not specific to remoting. You can use it at any time to serialize objects and store them. At the same time, it illustrates well how Windows PowerShell Remoting changes objects.

```powershell
PS> $original = Get-Process
PS> $original | Export-CliXML $home\persisted.xml
```

This command serializes the results from `Get-Process` and save them as XML in a file. This is precisely what Windows PowerShell Remoting does when it sends objects from a remote session back to the caller. Instead of storing the serialized objects in a file, Windows PowerShell Remoting transmits the XML to the caller via WinRM.

The file that `Export-CliXML` creates can be reimported and turned back into objects. This is often referred to as "rehydrating" the objects.

This is precisely what is happening when your local Windows PowerShell session receives serialized objects.

```powershell
PS> $clone = Import-CliXML $home\persisted.xml
```

When you compare `$original` and `$clone`, you notice that the number of objects did not change:

```powershell
PS> $original.count
59
PS> $clone.count
59
```

However, the object type changed. `Get-Member` proves that the objects in `$clone` no longer have methods:

```powershell
PS> $original | Get-Member -MemberType Method
```

<table>
<thead>
<tr>
<th>Name</th>
<th>MemberType Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginErrorReadLine</td>
<td>Method System.Void BeginErrorReadLine()</td>
</tr>
<tr>
<td>BeginOutputReadLine</td>
<td>Method System.Void BeginOutputReadLine()</td>
</tr>
<tr>
<td>CancelErrorRead</td>
<td>Method System.Void CancelErrorRead()</td>
</tr>
<tr>
<td>CancelOutputRead</td>
<td>Method System.Void CancelOutputRead()</td>
</tr>
<tr>
<td>Close</td>
<td>Method System.Void Close()</td>
</tr>
<tr>
<td>CloseMainWindow</td>
<td>Method bool CloseMainWindow()</td>
</tr>
<tr>
<td>Method</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CreateObjRef</td>
<td>System.Runtime.Remoting.ObjRef CreateObjRef(type ...)</td>
</tr>
<tr>
<td>Dispose</td>
<td>System.Void Dispose()</td>
</tr>
<tr>
<td>Equals</td>
<td>bool Equals(System.Object obj)</td>
</tr>
<tr>
<td>GetHashCode</td>
<td>int GetHashCode()</td>
</tr>
<tr>
<td>GetLifetimeService</td>
<td>System.Object GetLifetimeService()</td>
</tr>
<tr>
<td>GetType</td>
<td>type GetType()</td>
</tr>
<tr>
<td>InitializeLifetimeService</td>
<td>System.Object InitializeLifetimeService()</td>
</tr>
<tr>
<td>Kill</td>
<td>System.Void Kill()</td>
</tr>
<tr>
<td>Refresh</td>
<td>System.Void Refresh()</td>
</tr>
<tr>
<td>Start</td>
<td>bool Start()</td>
</tr>
<tr>
<td>ToString</td>
<td>string ToString(), string ToString(string format, System.IFormatProvider...)</td>
</tr>
<tr>
<td>WaitForExit</td>
<td>bool WaitForExit(int milliseconds), System.Void WaitForExit()</td>
</tr>
<tr>
<td>WaitForInputIdle</td>
<td>bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle()</td>
</tr>
</tbody>
</table>

```
PS> $clone | Get-Member -MemberType Method

TypeName: Deserialized.System.Diagnostics.Process

Name          MemberType Definition
------------- ---------------------------
ToString       Method               string ToString(), string ToString(string format, System.IFormatProvider)
```

Notice the first line that starts with `TypeName`. It gives you a clue as to just why the objects differ. The original objects returned by `Get-Process` were of type `System.Diagnostics.Process`. The “rehydrated” objects that were imported from XML have a type of `Deserialized.System.Diagnostics.Process`. The object types changed.

Not only did the serialization process remove all specific methods, it also changed some properties:

```
PS> $original | Get-Member -MemberType *Property* | Measure-Object | Select-Object -Expand Count 67
PS> $clone | Get-Member -MemberType *Property* | Measure-Object | Select-Object -Expand Count 49
```

Whereas the original object contained 67 properties, the rehydrated objects have 18 fewer properties. `Compare-Object` can show which properties were removed:

```
PS> $prop_orig = $original | Get-Member -MemberType *Property*
PS> $prop_clone = $clone | Get-Member -MemberType *Property*
PS> Compare-Object $prop_orig $prop_clone -Property Name
```

<table>
<thead>
<tr>
<th>Name</th>
<th>SideIndicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExitCode</td>
<td>&lt;=</td>
</tr>
<tr>
<td>ExitTime</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Handle</td>
<td>&lt;=</td>
</tr>
<tr>
<td>HasExited</td>
<td>&lt;=</td>
</tr>
<tr>
<td>MainModule</td>
<td>&lt;=</td>
</tr>
<tr>
<td>MaxWorkingSet</td>
<td>&lt;=</td>
</tr>
<tr>
<td>MinWorkingSet</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Modules</td>
<td>&lt;=</td>
</tr>
<tr>
<td>PriorityBoostEnabled</td>
<td>&lt;=</td>
</tr>
<tr>
<td>PriorityClass</td>
<td>&lt;=</td>
</tr>
<tr>
<td>PrivilegedProcessorTime</td>
<td>&lt;=</td>
</tr>
<tr>
<td>ProcessorAffinity</td>
<td>&lt;=</td>
</tr>
<tr>
<td>StandardError</td>
<td>&lt;=</td>
</tr>
<tr>
<td>StandardInput</td>
<td>&lt;=</td>
</tr>
<tr>
<td>StandardOutput</td>
<td>&lt;=</td>
</tr>
</tbody>
</table>
Windows PowerShell Remoting works very similarly, but not the same way. To compare original objects with deserialized objects received from Windows PowerShell Remoting, you can use a cmdlet that supports remote access internally - like Get-Process. When you run Get-Process with the ComputerName parameter, it uses traditional DCOM technologies to deliver the real objects. When you use Invoke-Command to run Get-Process via Windows PowerShell Remoting, you receive deserialized objects. All you need to do is compare the results.

PS> $original = Get-Process -ComputerName storage1
PS> $clone = Invoke-Command { Get-Process } -ComputerName storage1

The first observation is speed: the second call using Windows PowerShell Remoting is much slower, and you now know why: first Windows PowerShell Remoting needs to instantiate a remote session on the target system. Then the results need to be converted into CliXML format similar to Export-CliXML, which takes considerable time. Then the XMLized objects need to travel back to the caller and be reimported and turned back into objects. This is the price (overhead) you pay for Windows PowerShell Remoting. While the first call took 0,14 seconds, the second call required 10,8 seconds, so it was approximately 80 times slower.

PS> Measure-Command { $original = Get-Process -ComputerName storage1 }

Days  : 0
Hours : 0
Minutes: 0
Seconds: 0
Milliseconds: 141
Ticks : 1410010
TotalDays : 1,63195601851852E-06
TotalHours : 3,91669444444444E-05
TotalMinutes : 0,0023500166666667
TotalSeconds : 0,141001
TotalMilliseconds : 141,001

PS> Measure-Command { $clone = Invoke-Command { Get-Process } -ComputerName storage1 }

Days  : 0
Hours : 0
Minutes: 0
Seconds: 0
Milliseconds: 812
Ticks : 108124609
TotalDays : 0,00012514422337963
TotalHours : 0,0030034613611111
TotalMinutes : 0,180207681666667
TotalSeconds : 10,8124609
TotalMilliseconds : 10812,4609

When you analyze object properties, you will discover that Windows PowerShell Remoting not only removed some properties but also added others:

PS> $prop_orig = $original | Get-Member -MemberType *Property*
PS> $prop_clone = $clone | Get-Member -MemberType *Property*
PS> Compare-Object $prop_orig $prop_clone -Property Name

<table>
<thead>
<tr>
<th>Name</th>
<th>SideIndicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSComputerName</td>
<td>=&gt;</td>
</tr>
</tbody>
</table>
The added properties help you to determine where the objects came from and in which session they were running, which becomes important when you invoke commands simultaneously in multiple sessions.

**Windows PowerShell Sessions are Fragile**

Windows PowerShell Remoting relies on sessions, and a session represents a running Windows PowerShell instance. Therefore, a session isn't just a plain network connection that you can restore easily. Instead, it is a room where Windows PowerShell lives and you can configure a session, load commands, and define functions and variables inside it. That is why Windows PowerShell sessions are unique and cannot automatically be restored easily.

This can be a problem because sometimes sessions end unexpectedly. Because they represent a running Windows PowerShell host, they are closed, like any other program, when a remote system reboots. Rebooting a remote system will, therefore, end all current Windows PowerShell Remoting sessions running on that system. These connections are not restored when the system finishes booting.

The impact of this depends on what you are doing. If all you want to do is fire a remote command to a remote system, then a session lifespan is short and chances are low that this session will break. If, however, you are using implicit remoting to import remote commands that you plan to use in your local session over a long period of time, then a reboot of the target machine may break your environment unexpectedly. Imported commands do remember the session they were created in, so when you use such a command and the original session is lost, there will be an attempt to recreate the session. Since it is no longer the original session, some adjustments you may have made to it will be lost.

**Background Jobs**

Remoting and background jobs share the same concept and complement each other to some degree which is why background jobs are covered as well in this paper.

A background job is composed of a separate session that runs parallel to your local session to complete tasks asynchronously. Because Windows PowerShell is single-threaded, it can do things only sequentially, and when tasks are complex, you might have to wait until the results come in. With background jobs, you transfer the job to another session and leave it to this session to process and complete it. This way, your own session is not blocked. You can check on jobs and receive the results whenever it suits you.

**Running Jobs**

To better understand how background jobs work, look at a simple example. The next command finds all log files in your Windows folder and its subfolders, which can take considerable time:

```
PS> Dir $env:windir *.log -Recurse
```

While the command runs, your local session is blocked. To execute the same task asynchronously, simply start a new session, and transfer the task to that session. This is done by *Start-Job*:

```
PS> Start-Job {Dir $env:windir *.log -Recurse}
```

The job now runs in the new session that was created transparently by *Start-Job*. Your own session is neither blocked nor busy working on the task you transferred to the other session. Your own session keeps a connection to the background job session so you can check on the job status and receive its results whenever you feel like it.

To check the status of jobs, use *Get-Job*:
This tells you if a particular job is done and has gathered data that is ready to collect. To receive the results a job has gathered, you call `Receive-Job`:

```
PS> Receive-Job -Id 3
```

Once you received data from a job, this data is removed from the job buffer (unless you specify the `Keep` switch parameter). When a job is complete and you have received all of its results, it is time to clean up and remove the job:

```
PS> Remove-Job -Id 3
```

As it turns out, background jobs and remoting code both employ additional Windows PowerShell sessions and provide ways for you to communicate with them. They share the same concept but serve different purposes:

**Remoting:** Remoting code runs in a separate session to enable you to move the place of execution to other systems. The remote sessions run synchronously, though. Your local session is blocked as long as the remote sessions execute, and when done, the results are transferred back to your local session using WinRM and the XML packaging process you read about.

**Background Jobs:** Job code runs in a separate session to enable you to execute tasks in parallel and separately from your local session. Your local session is not blocked while jobs execute. The results are buffered by the job sessions and are transferred back to your local session only when you are ready for them and request them through `Receive-Job`. The background sessions uses the same XML packaging process to transfer the results to your local session that is used by remoting.

### Creating Remote Jobs

`Invoke-Command` is used to create new sessions that can run remotely, but these sessions run synchronously and block your own session. `Start-Job` is used to create new sessions that can run asynchronously, but you can only create these sessions on your local machine. Is there a way to connect both concepts?

`Start-Job` supports only local jobs because this cmdlet was designed to run independently from remoting requirements. You can run background jobs without having to enable Windows PowerShell Remoting. However, `Invoke-Command` has a way of creating remote sessions as jobs. Simply use the `AsJob` switch parameter to create a remote session that runs asynchronously:

```
PS> Invoke-Command { $env:windir *.log -Recurse } -ComputerName PC01 -AsJob
```

This command returns immediately and returns a job object. The remoting task executes asynchronously, and you can use `Get-Job` and `Receive-Job` to check its status and collect its results whenever it suits you.

There are many cmdlets that support the `AsJob` switch parameter and can create asynchronous background jobs on the fly. To find them, look for cmdlets with a `AsJob` parameter:

```
PS> Get-Help * -Parameter AsJob
```

### Troubleshooting

All remoting technologies are governed by a number of network and security settings. It is not always easy to understand what is required and why a remoting call might have failed.

**Windows PowerShell Remoting**

In a perfect world, setting up and running Windows PowerShell Remoting would be easy and straightforward. Unfortunately, any form of remote access is influenced by a flood of security rules and settings. If your system is not set up with default settings, or if security has been hardened, you may experience difficulties setting up and running Windows PowerShell Remoting.
I cannot set up Windows PowerShell Remoting!

When you try and set up Windows PowerShell Remoting by running `Enable-PSRemoting`, you might be faced with a number of error messages. Let's see what might cause them and what you can do about it.

- **Access Denied**: To set up Windows PowerShell Remoting, local Administrator privileges are required. You need to start Windows PowerShell with elevated privileges. To do so, right-click any Windows PowerShell icon or shortcut, and then click “Run as Administrator”.

- **Public Networks Detected**: Part of the setup procedure is setting up a local firewall exception. If `Enable-PSRemoting` detects that your system is connected to a public network, it fails, and all subsequent setup tasks are discarded. Start `Network and Sharing Center` and make sure that all public connections are either Private or Domain type networks or are disabled.

  `Enable-PSRemoting` also might fail when the predefined firewall rules for WinRM have been deleted. To recreate these rules, run `netsh advfirewall reset`.

- **Access Denied Despite Admin Privileges**: If you are logged on as a user with local Administrator privileges and you start Windows PowerShell with Administrator privileges, and you still get Access Denied errors, log with the built-in Administrator account. Run `control userpasswords2`, click `Advanced` and make sure the `Administrator` account is enabled. Then start Windows PowerShell with this account and try:

  ```powershell
  PS> Runas /user:Administrator powershell
  ```

- **Invalid SSIDs**: If you receive an error that complains about invalid SSIDs in XML data, there might be leftover SSIDs in the Windows Registry that refer to accounts that no longer exist. Open the registry editor, and navigate to this location:

  `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WSMAN\Plugin`  

Or start Windows PowerShell with the “Run as Administrator” option and type:

```powershell
PS> Get-ChildItem \wsman:\localhost\plugin
```

Using the WSMAn: drive is safer than munging with the registry directly, but if permission issues prevent you from accessing WSMAn: from inside PowerShell, direct registry cleanup may be required.

Make sure there are no subkeys other than `Event Forwarding Plugin`, `microsoft.powershell`, `microsoft.powershell32`, and `WMI Provider`. `microsoft.powershell` and `microsoft.powershell32` are predefined session configurations created by `Enable-PSRemoting`. `microsoft.powershell32` is present only on 64-bit systems and represents the 32-bit Windows PowerShell host. As a last resort, you can delete these two and call `Enable-PSRemoting` to recreate these configurations from scratch.

I cannot create a Session!

To make sure Windows PowerShell Remoting is set up correctly on your own system, first try to create a session to your own computer. To do that, you must be a local Administrator and you must start Windows PowerShell with the “Run as Administrator” option.

```powershell
PS> $session = New-PSSession -ComputerName localhost
```

If this fails, Windows PowerShell Remoting may not be set up correctly. Make sure that `Enable-PSRemoting` ran correctly, and also make sure you adjusted the `TrustedHosts` list.

Next, try to create a session on a different system:

```powershell
PS> $session = New-PSSession -ComputerName PC01
```

If this works, you are fine. If you receive an error message, check the error message:

- **RPC Server unavailable**: Your computer was unable to contact the target system. Either, the system is offline, or you may have mistyped its name or IP address. If the computer name or IP address is correct and the system is online, a firewall probably blocks access. Make sure you have set up Windows PowerShell Remoting correctly on the remote system.

- **Access Denied**: This is actually a good message because it proves that you successfully connected to the target system. You just did not have access permission. When you do not use the Credential parameter, you are authenticated as the currently logged-on user. Make sure you know the user name and password for an account that has local Administrator privilege, and then use `Credential`
authenticate as this user. If you provided the credentials of a user that is definitely member of the local Administrators group, and you still get Access Denied, temporarily disable the firewall. If access works now, the firewall blocks important ports. Run Enable-PSRemoting again and make sure the firewall exception is set up correctly.

- **Public Network:** Always remember that all firewall exceptions work only with Private and Domain network profiles. If your network is classified as Public network in Network and Sharing Center, other systems cannot access your computer.

- **No Kerberos:** When you receive lengthy error messages that complain about WinRM not being able to authenticate you unless you use Kerberos or HTTPS: transports, you know that Kerberos is not available for this particular call. Most likely, you are not a domain member or you are trying to connect to a system outside your own domain. Kerberos is also not available when you specify a remote computer by using its IP address instead of computer name, or when you have no connection to your domain controller. To connect under these conditions, make sure you set the TrustedHosts list to »*« then as described earlier, and try again. You might have to use the Credential parameter to supply appropriate credentials.

**I want to check my remoting settings manually:**

To make sure all prerequisites for Windows PowerShell Remoting are in place, check them manually:

- **WinRM Service:** This service needs to be running. It is handling the network transport. To check the status, use Get-Service WinRM. Next, test your own side:

  ```powershell
  PS> Test-WSMan -Authentication default
  ``

  This should return a quick report that includes your operating system and version. If not, WinRM is not set up correctly on your machine.

  Next, check the destination computer:

  ```powershell
  PS> Test-WSMan -ComputerName PC01 -Authentication default
  ``

  This should return a similar report from the remote system. If not, WinRM is not set up correctly on the remote machine.

- **Listener required:** To receive remoting requests, WinRM needs a listener on port 5985. To check your listeners,

  ```powershell
  PS> Get-WSManInstance -ResourceURI winrm/config/listener -Enumerate | Select port, transport, ListeningOn
  ``

<table>
<thead>
<tr>
<th>Port</th>
<th>Transport</th>
<th>ListeningOn</th>
</tr>
</thead>
<tbody>
<tr>
<td>5985</td>
<td>HTTP</td>
<td>10.10.10.2, 127.0.0.1, 16...</td>
</tr>
</tbody>
</table>

  You should discover at least one listener on port 5985 and transport HTTP. If not, Windows PowerShell Remoting is not set up correctly.
Firewall-Exception: When you set up Windows PowerShell Remoting, it adds a firewall exception to your local Windows firewall that allows other computers to send requests to port 5985. If this exception is not present, your computer is unreachable, and you should run Enable-PSRemoting again to (re)create the exception. Note that this firewall exception is not active in your Public network profile, so you may also want to check your current network type in Network and Sharing Center and change it to something other than Public. Use Get-FirewallRule, as described earlier in this paper, to check your firewall rules.

Non-Windows PowerShell Remoting

Problems with cmdlets that have built-in remoting capabilities are harder to diagnose because there is no common transport. Still, they share a number of prerequisites with Windows PowerShell Remoting:

- RPC Server unavailable: As with Windows PowerShell Remoting, this error indicates a connectivity problem or a missing firewall exception. You might also want to test the physical network connection:

  PS> Test-Connection PC01

  Try pinging the target system, both with its computer name and its IP address. If the IP address works but the computer name fails, you might have a problem with name resolution in your network infrastructure.

  If you do not receive a signal from the target system at all, this does not necessarily indicate a non-existing (offline) system. It also can be caused by inappropriate firewall settings. By default, enabling the Windows PowerShell Remoting firewall exception also enables ICMP, so if Test-Connection fails, this indicates either a non-available system or blocking firewall(s).

  Turning off the Windows firewall temporarily on both ends can help diagnose the problem. If this solves the connectivity problem, you know there is a need for some firewall exception. Do not turn off the Windows firewall permanently. It is an important part of your security system. Instead, make sure the firewall is configured correctly.

  Always make sure the target system network type is not set to Public because, when it is, all mandatory firewall exceptions are disabled.

- Access Denied: Your current user account does not have local Administrator privileges on the target machine or you are logged on with a local account but the target machine is domain-joined. If the cmdlet you are using has a Credential parameter, try it with a different user account. If the user account used does have local Administrator privileges on the target system, temporarily disable the Windows firewall. If that solves the problem, you know that firewall exceptions are missing.

- Other errors: There might be additional prerequisites needed on the target machine, depending on the cmdlet you choose and the technology it uses to access the target system. For example, to use Get-EventLog with its ComputerName parameter to access event log entries remotely, the target system must run the RemoteRegistry service; otherwise you receive confusing error messages.

Summary

Windows PowerShell 2.0 essentially adds the ability for sessions to communicate with each other. This enables you to run any Windows PowerShell code either remotely or as a background job - or both.

To create a session on remote systems, use New-PSSession, Invoke-Command or Enter-PSSession with its ComputerName property. To create a session for local background jobs, use Start-Job or use other cmdlets like Get-WmiObject which have the AsJob parameter. For example, to create a remote session as background job, use Invoke-Command with the AsJob parameter.

Windows PowerShell Remoting essentially works like a web service, so it does not need any special setup for the client (caller). Only the server (target) needs to run the WinRM service with a listener and a firewall exception. So, on the server side, make sure you run Enable-PSRemoting first.
On the caller side, you are, by default, limited to Kerberos authentication. If you need to authenticate differently, you need to add the target computers to a *TrustedHosts* list that is maintained by WinRM. This is why, in a peer-to-peer scenario, you need to enable Windows PowerShell Remoting on the client side as well and then add »*« to the *TrustedHosts* setting.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>PS Remoting Setup</th>
<th>TrustedHosts List</th>
<th>Full Admin Privileges for PS Console</th>
<th>Network Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-Domain with computername</td>
<td>on target side</td>
<td>not necessary</td>
<td>not necessary</td>
<td>non-public on target side</td>
</tr>
<tr>
<td>Domain-Domain with IP address</td>
<td>on both sides</td>
<td>necessary</td>
<td>not necessary</td>
<td>non-public on both sides</td>
</tr>
<tr>
<td>Domain-Nontrusted Domain</td>
<td>on both sides</td>
<td>necessary</td>
<td>not necessary</td>
<td>non-public on both sides</td>
</tr>
<tr>
<td>Peer-Peer</td>
<td>on both sides</td>
<td>necessary</td>
<td>not necessary</td>
<td>non-public on both sides</td>
</tr>
<tr>
<td>Peer-Domain</td>
<td>on both sides</td>
<td>necessary</td>
<td>not necessary</td>
<td>non-public on both sides</td>
</tr>
<tr>
<td>local</td>
<td>enabled</td>
<td>necessary</td>
<td>necessary</td>
<td>non-public</td>
</tr>
</tbody>
</table>

*Table 3: Requirements for different types of Windows PowerShell Remoting*

Receiving results from a session is done by temporarily converting objects to *CliXML*. Any object you receive from a remote or background job session is really a clone with only a subset of properties and no methods. Sending back results is the most expensive part of remoting, so you should try to limit the information that needs to be returned to you, and try to process objects on the server side for as long as possible.

Classic remoting that is built into individual cmdlets, such as *Get-WmiObject*, is a completely different ball game and is not related to Windows PowerShell sessions or Windows PowerShell Remoting. This type of remoting returns original objects because it uses DCOM instead of XML serialization. Since it requires less overhead, it can be much faster and does not require Windows PowerShell 2.0 on the target side. However, it is only supported by a subset of cmdlets, uses different kinds of different technologies internally to access systems remotely. Whether classic remoting works for you depends on a number of prerequisites on client and server and is beyond the scope of this paper.
PowerShell Plus

Architected and developed by Dr. Tobias Weltner, Idera’s PowerShell Plus is the most advanced Interactive Development Environment for Windows PowerShell available today. Designed to help administrators and developers quickly learn and master Windows PowerShell, it also dramatically increases the productivity of expert users. Try PowerShell Plus free for 30-days and see how much more productive you can be!

PowerShell Plus Interactive Console
Enables you to work interactively with Windows PowerShell from a feature rich Windows UI. This integration makes working with Windows PowerShell faster and easier to use than ever before!

Advanced Script Editor
The advanced debugger and script editor lets you build and test complex Windows PowerShell scripts, try one line Windows PowerShell commands from an embedded console window, and sign your script with a security certificate… All from a single workspace!

Comprehensive Learning Center
The Comprehensive Learning Center helps you experience Windows PowerShell by example. Short tutorials guide you through basic concepts at your own pace. The Comprehensive Learning Center also includes dynamically created help topics from currently installed Windows PowerShell cmdlets, snap-ins and WMI objects.

» Try PowerShell Plus Free for 30-Days
PowerShell.com Sponsors

Idera

Idera delivers a new generation of tools for managing, administering, and securing Microsoft Windows Servers. Idera's products help companies ensure server performance and availability and reduce administrative overhead and expense. Idera provides solutions for SQL Server, SharePoint and Windows PowerShell. Headquartered in Houston, Texas, Idera’s products are sold and supported directly and via authorized resellers and distributors around the globe. To learn more, please contact Idera at +1-713.523.4433 or visit www.idera.com.

Compellent

Compellent is a leading provider of enterprise-class network storage solutions that are highly scalable, feature-rich and designed to be easy to use and cost effective. Compellent Technologies’ principal offices are located in Eden Prairie, Minn. www.compellent.com/powershell.

/n software

/n software is a leading provider of software components for Internet, security, and E-Business development. Founded in 1994, /n software (pronounced ‘n software’) is based in Research Triangle Park, North Carolina. You can reach the company via email at info@nsoftware.com, on the World Wide Web at www.nsoftware.com or by calling US: (800) 225-4190 or International: (919) 544-7070.