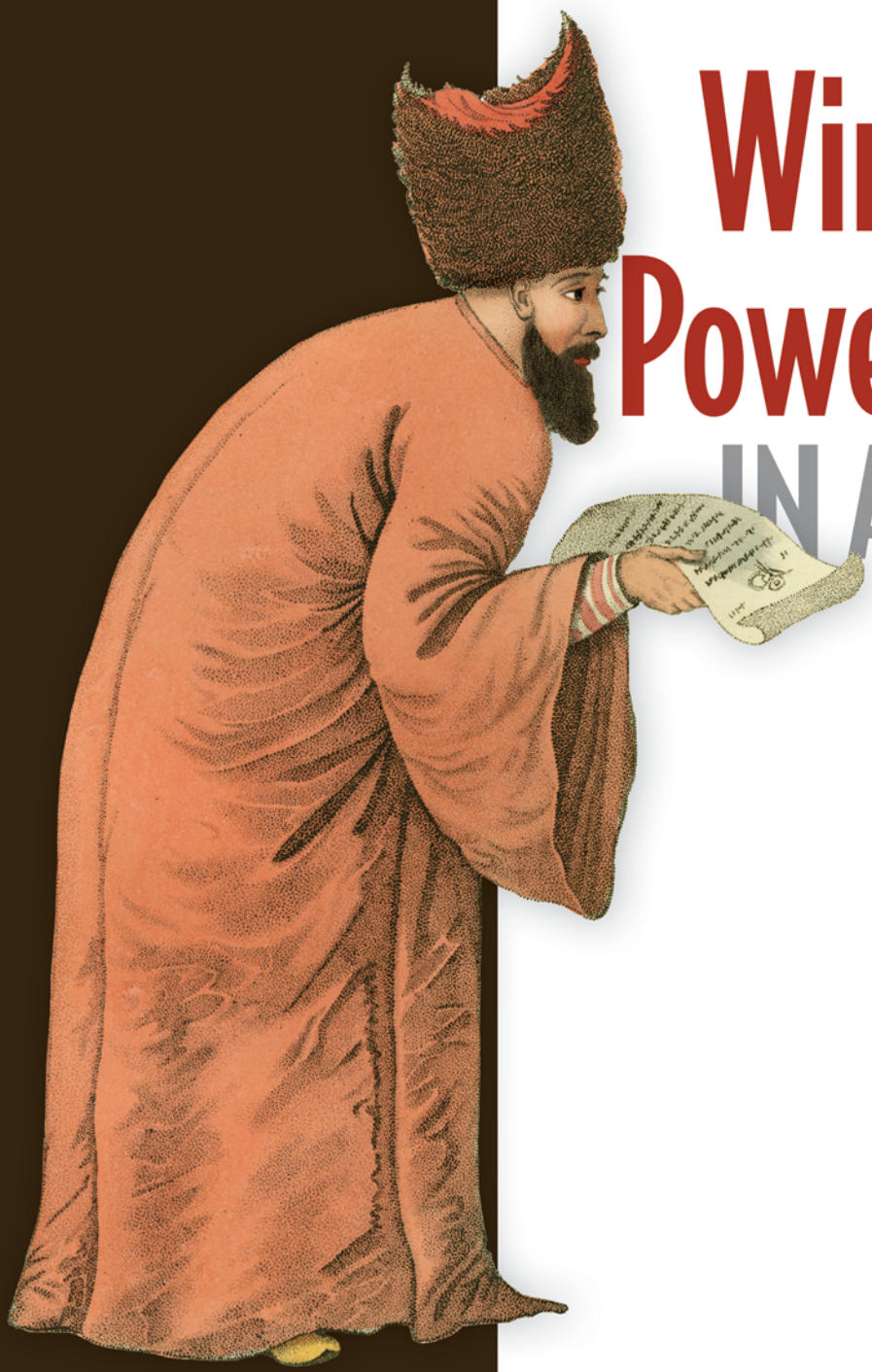
SAMPLE CHAPTER

THIRD EDITION



Windows PowerShell IN ACTION

Bruce Payette
Richard Siddaway

MANNING

Windows PowerShell in Action
Third Edition

by Bruce Payette
Richard Siddaway

Chapter 11

brief contents

11 PowerShell remoting

This chapter covers

- Commands with built-in remoting
- PowerShell remoting subsystem
- Using PowerShell remoting
- Remoting sessions, persistent connections, and implicit remoting
- Remoting considerations and custom remoting sessions

In a day when you don't come across any problems, you can be sure that you are traveling in the wrong path.

—Swami Vivekananda

PowerShell is a tool intended for enterprise and cloud management but if it can't manage distributed systems it isn't useful. Fortunately, PowerShell has a comprehensive built-in remoting subsystem. This facility allows you to handle most remoting tasks in any kind of configuration you might encounter.

In this chapter, we're going to cover the features of remoting and how you can apply them. We'll use an example showing how to combine the features to solve a nontrivial problem: monitoring multiple remote machines. We'll then look at

some of the configuration considerations you need to be aware of when using PowerShell remoting.

Let's start with a quick overview of PowerShell remoting.

11.1 PowerShell remoting overview

The ultimate goal for remoting is to be able to execute a command on a remote computer. There are two ways to approach this. First, you could have each command do its own remoting. In this scenario, the command is still executed locally but uses system-level networking capabilities like DCOM to perform remote operations. A number of commands do this, which we'll cover in the next section. The negative aspect of this approach is that each command has to implement and manage its own remoting and authentication mechanisms.

PowerShell includes a second, more general solution, allowing you to send a command (or pipeline of commands or even a script) to the target machine for execution and then retrieve the results. With this approach, you only have to implement the remoting mechanism once and then it can be used with any command. This second solution is the one we'll spend most of our time discussing. But first let's look at the commands that implement their own remoting.

11.1.1 Commands with built-in remoting

A number of commands in PowerShell have a `-ComputerName` parameter, which allows you to specify the target machine to access. You can discover (some of) these cmdlets by running either of these commands:

```
PS> Get-Help * -Parameter ComputerName
PS> Get-Command -ParameterName ComputerName
```

CIM sessions

Common Information Model (CIM) sessions (see chapter 16) are closely related to PowerShell remoting—they enable more efficient access to WMI classes on remote machines. The cmdlets capable of using CIM sessions can be discovered in a similar way:

```
PS> Get-Command -ParameterName Cimsession
```

For a new PowerShell v5.1 session on Windows 10, the majority of the cmdlets are listed in table 11.1.

NOTE The number of cmdlets you see will depend on the modules you have on your machine. You won't necessarily see identical results from the two commands given earlier because `Get-Help` is dependent on analyzing the help files.

Table 11.1 Cmdlets with built-in remoting capability

Add-Computer	Clear-EventLog	Connect-PSSession
Enter-PSSession	Get-EventLog	Get-HotFix
Get-Process	Get-Service	Get-WmiObject
Invoke-Command	Invoke-WmiMethod	Limit-EventLog
New-EventLog	New-PSSession	Receive-Job
Receive-PSSession	Register-WmiEvent	Remove-Computer
Remove-EventLog	Remove-PSSession	Remove-WmiObject
Rename-Computer	Restart-Computer	Send-MailMessage
Set-DscLocalConfigurationManager	Set-Service	Set-WmiInstance
Show-EventLog	Start-DscConfiguration	Stop-Computer
Test-Connection	Write-EventLog	

NOTE We've deliberately excluded the `*WSMan*` cmdlets from table 11.1. The `*WSMan*` cmdlets are effectively deprecated and have been replaced by the `*-CIM*` cmdlets (see chapter 16).

These commands do their own remoting because either the underlying infrastructure already supports remoting or they address scenarios that are of particular importance to system management. You need to supply only one or more computer names to use them against a remote target:

```
PS> Get-Service -Name BITS -ComputerName W16TGT01, W16DSC02

 Status Name DisplayName
 ------ ---- -----------
Stopped BITS Background Intelligent Transfer Service
Stopped BITS Background Intelligent Transfer Service
```

You don't get any indication of which result belongs to which machine by default. In this case, you need to include the `MachineName` property in the output:

```
PS> Get-Service -Name BITS -ComputerName W16TGT01, W16DSC02 |
select Status, Name, MachineName

Status Name MachineName
 ------ ---- -----------
Stopped BITS W16DSC02
Stopped BITS W16TGT01
```

Self-remoting is performed using DCOM and RPC. These protocols will be blocked by default by firewalls. Also, the set of commands that do self-remoting is quite small, so the remaining commands must rely on the PowerShell remoting subsystem to access remote computers. We'll start looking at that in the next section.

11.1.2 The PowerShell remoting subsystem

You've seen a few brief examples of how remoting works in previous chapters. You may remember that all those examples used the same basic cmdlet: `Invoke-Command`. This cmdlet allows you to remotely invoke a scriptblock on another computer and is the building block for most of the features in remoting. The partial syntax for this command is shown in figure 11.1.

```
Invoke-Command [[-ComputerName] <string[]>] [-ScriptBlock] <scriptblock>
[-Credential <pscredential>] [-Port <int>] [-UseSSL]
[-ConfigurationName <string>] [-ApplicationName <string>]
[-ThrottleLimit <int>] [-AsJob] [-InDisconnectedSession]
[-SessionName <string[]>] [-HideComputerName] [-JobName <string>]
[-SessionOption <PSSessionOption>]
[-Authentication <AuthenticationMechanism>] [-EnableNetworkAccess]
[-InputObject <psobject>] [-ArgumentList <Object[]>]
[-CertificateThumbprint <string>] [<CommonParameters>]
[-VMId] <guid[]> -VMName <string[]> -ContainerId <string[]>
[[-ConnectionUri] <uri[]>]
```

Figure 11.1 Partial syntax for the `Invoke-Command` cmdlet, which is the core of PowerShell's remoting capabilities. This cmdlet is used to execute commands and scripts on one or more computers. It can be used synchronously or asynchronously as a job. The `VMId`, `VMName`, and `ContainerId` parameters were introduced with PowerShell 5.1 and are valid only on Windows 10 and Windows Server 2016 (or later).

The `Invoke-Command` cmdlet is used to invoke a scriptblock on one or more computers. You do so by specifying a computer name (or list of names) for the machines on which you want to execute the command. For each name in the list, the remoting subsystem will take care of all the details needed to open the connection to that computer, execute the command, retrieve the results, and then shut down the connection. If you're going to run the command on a large set of computers, `Invoke-Command` will also take care of all resource management details, such as limiting the number of concurrent remote connections. Our previous example becomes this:

```
PS>  Invoke-Command -ScriptBlock {Get-Service -Name BITS} `
-ComputerName W16TGT01, W16DSC02

Status  Name DisplayName                                PSComputerName
------  ---- -----------                                --------------
Stopped BITS Background Intelligent Transfer Service W16DSC02
Stopped BITS Background Intelligent Transfer Service W16TGT01
```

Note that you now get the computer name that the result refers to in the output.

This is a simple but powerful model if you need to execute only a single command or script on the target machine. But if you want to execute a series of commands on the target, the overhead of setting up and taking down a connection for each command becomes expensive. PowerShell remoting addresses this situation by allowing you to

create a persistent connection to the remote computer called a *session*. You do so by using the `New-PSSession` cmdlet.

Both of the scenarios we've discussed so far involve what is called *noninteractive remoting* because you're only sending commands to the remote machines and then waiting for the results. You don't interact with the remote commands while they're executing.

Another standard pattern in remoting occurs when you want to set up an *interactive session* where every command you type is sent transparently to the remote computer. This is the style of remoting implemented by tools like Remote Desktop, Telnet, or SSH (Secure Shell).

> **NOTE** The PowerShell team has announced that SSH support will be built into PowerShell. Basic terminal support will be available with Windows Server 2016. Full SSH integration with the PowerShell Remoting Protocol will be introduced at a later date. Appendix A demonstrates SSH-based remoting between Linux and Windows machines using PowerShell v6.

PowerShell allows you to start an interactive session using the `Enter-PSSession` cmdlet. Use `Exit-PSSession` to close the session when you've finished working. If you enter a remote session created by `New-PSSession`, then using `Exit-PSSession` will suspend the session without closing the remote connection. Because the connection isn't closed, you can later reenter the session with all session data preserved by using `Enter-PSSession` again. An example of an interactive session is given in figure 11.2.

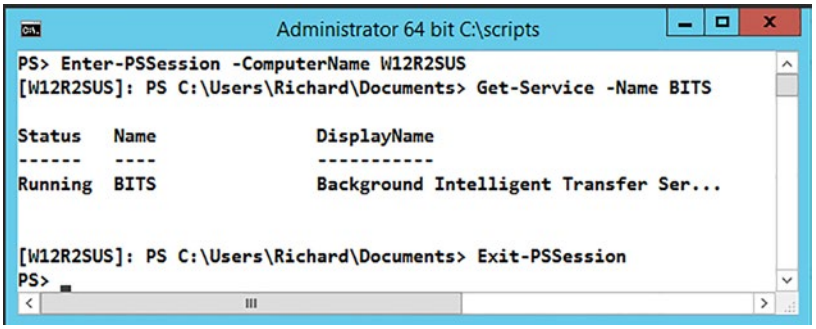


Figure 11.2 Interactive remoting session to the computer W12R2SUS. Notice how the PowerShell prompt changes to incorporate the remote machine name when you enter the session.

These cmdlets—`Invoke-Command`, `New-PSSession`, and `Enter-PSSession`—are the basic remoting tools you'll be using. But before you can use them, you need to make sure remoting is enabled, so we'll look at that next.

11.1.3 Enabling remoting

At this point we have some good news and some bad news for you. The good news is that for Windows Server 2012 and later (including Windows Server 2012 R2 virtual machines running in Azure IaaS), PowerShell remoting is enabled by default. The bad news is that

for earlier versions of Window Server and for all versions of the Windows client operating system, PowerShell remoting is turned off by default and has to be enabled.

> **NOTE** You have to turn on PowerShell remoting for a machine to receive and execute remote administration commands. You don't need to turn on remoting to send commands, though you will need to turn it on at least temporarily to change client-side settings such as the `TrustedHosts` list on the local machine.

You enable remoting using the `Enable-PSRemoting` cmdlet. To run this command, you must have administrator privileges on the machine you're going to enable. You need to do the following:

- Start the PowerShell session with elevated privileges (Run As Administrator).
- Ensure that none of the network connections on the machine has a network profile of `Public`. Use `Get-NetConnectionProfile | Set-NetConnectionProfile -NetworkCategory Private` to set the network profile.

By default, `Enable-PSRemoting` runs silently with no output and no input required. You can use the `-Verbose` and `-Confirm` parameters to see what's happening, as shown in figure 11.3.

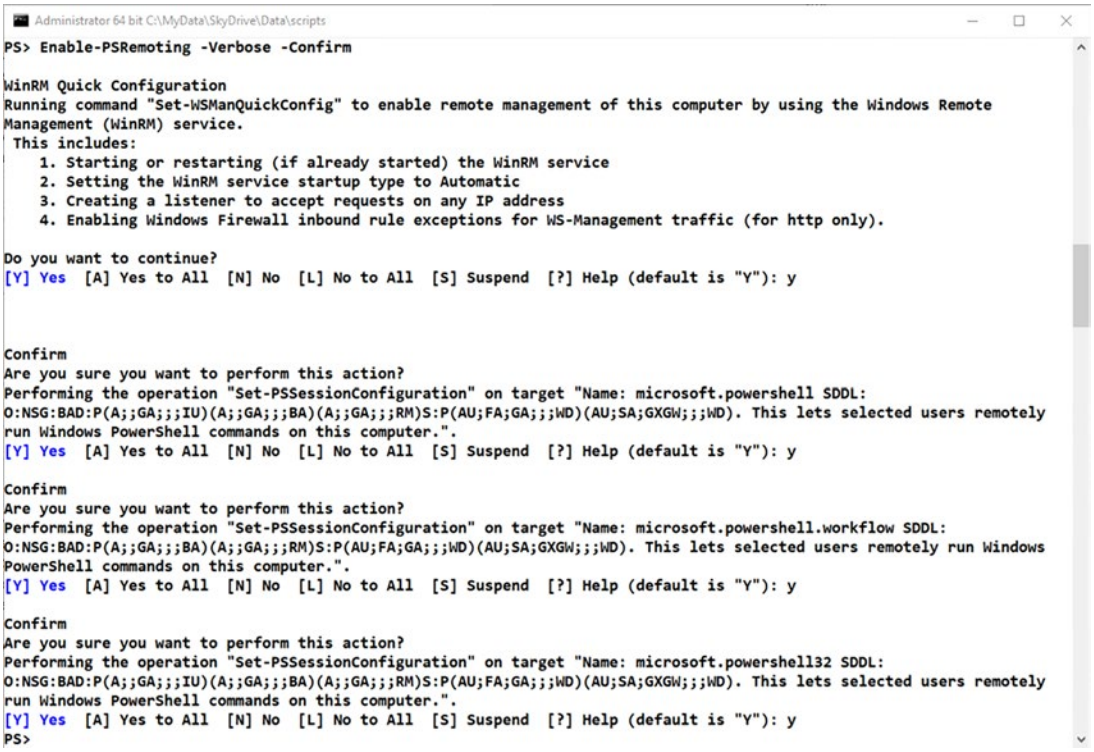


Figure 11.3 Enabling PowerShell remoting on a machine

The `Enable-PSRemoting` command performs all the configuration steps needed to allow users with local administrator privileges to remote to this computer in a domain environment. In a non-domain or workgroup environment, as well as for non-admin users, additional steps are required for remoting to work.

11.1.4 Additional setup steps for workgroup environments

If you're working in a workgroup environment—for example, at home—you must take a few additional steps before you can connect to a remote machine. With no domain controller available to handle the various aspects of security and identity, you have to manually configure the names of the computers you trust. If you want to connect to the computer `computerItrust`, then you have to add it to the list of trusted computers (or `TrustedHosts` list).

You can do this via the `WSMan:` drive, as shown in table 11.2. Note that you need to be running as administrator to be able to use the `WSMan:` provider. Once you've completed these steps, you're ready to start playing with some examples.

Table 11.2 Additional steps needed to enable remote access to a computer in a workgroup environment

Step	Command	Description
1	`cd wsman:\localhost\client`	`cd`'ing into the client configuration node in the `WSMan:` drive allows you to access the WS-MAN configuration for this computer using the provider cmdlets.
2	`$old = (Get-Item .\ TrustedHosts).Value`	You'll want to update the current value of the `TrustedHosts` item, so you get it and save the value in a variable.
3	`$old += ',computerItrust'`	The value of `TrustedHosts` is a string containing a comma-separated list of the computers considered trustworthy. You add the new computer name to the end of this list, prefixed with a comma. (If you're comfortable with implicitly trusting any host, then set this string to `*`, which matches any hostname.)
4	`Set-Item .\TrustedHosts $old`	Once you've verified that the updated contents of the variable are correct, you assign it back to the `TrustedHosts` item, which updates the configuration.

A note on security

The computers in the `TrustedHosts` list are implicitly trusted by the *local* computer when you add their names to this list. It's *not* an incoming security feature like a firewall. The identity of these computers won't be authenticated when you connect to them. Because the connection process requires sending *credential information* to these machines, you need to be sure that you can trust these computers. Also, be aware that the `TrustedHosts` list on a machine applies to everyone who uses that computer, not only the user who changed the setting.

> ***(Continued)***
> That said, unless you allow random people to install computers on your internal network, this shouldn't introduce substantial risk most of the time. If you're comfortable with knowing which machines you'll be connecting to, you can put `*` in the `TrustedHosts` list, indicating that you're implicitly trusting any computer you might be connecting to. As always, security is a principle tempered with pragmatics.

An alternative way of validating the identity of the target computer is to use HTTPS when connecting to that computer. This works because, in order to establish an HTTPS connection, the target server must have a valid certificate installed where the name in the certificate matches the server name. As long as the certificate is signed by a trusted certificate authority you know that the server is the one it claims to be. Unfortunately, this process does require that you have a valid certificate, issued by either a commercial or local CA. This is an entirely reasonable requirement in an enterprise environment but may not always be practical in smaller or informal environments.

11.1.5 Authenticating the connecting user

In the previous section, you saw how the client verifies the identity of the target computer. Now we'll explore the converse of this—how the target computer verifies the identity of the connecting user. PowerShell remoting supports a wide variety of ways of authenticating a user, including NTLM and Kerberos. Each mechanism has its advantages and disadvantages. The authentication mechanism also has an important impact on how data is transmitted between the client and the server. Depending on how you authenticate to the server, the data passed between the client and server may or may not be encrypted. Encryption is extremely important in that it protects the contents of your communications with the server against tampering and preserves privacy. If encryption isn't being used, you need to ensure the physical security of your network. No untrusted access to the network can be permitted in this scenario. The possible types of authentication are shown in table 11.3.

Table 11.3 Possible types of authentication available for PowerShell remoting

Auth type	Description	Encrypted payload
Default	Use the authentication method specified by the WS-Management Protocol.	Depends on what was specified.
Basic	Use Basic Authentication, part of HTTP, where the username and password are sent unencrypted to the target server or proxy.	No. Use HTTPS to encrypt the connection.
Digest	Use Digest Authentication, which is also part of HTTP. This mechanism supersedes Basic Authentication and encrypts the credentials.	Yes.

Table 11.3 Possible types of authentication available for PowerShell remoting *(continued)*

Auth type	Description	Encrypted payload
Kerberos	The client computer and the server mutually authenticate using the Kerberos network authentication protocol.	Yes.
Negotiate	Negotiate is a challenge-response scheme that negotiates with the server or proxy to determine the scheme to use for authentication. For example, negotiation is used to determine whether the Kerberos protocol or NTLM is used.	Yes.
CredSSP	Use Credential Security Service Provider (CredSSP) authentication, which allows the user to delegate credentials. This mechanism, introduced with Windows Vista, is designed to support the second-hop scenario, where commands that run on one remote computer need to hop to another computer to do something.	Yes.

For all the authentication types except Basic, the payload of the messages you send is encrypted directly by the remoting protocol. If Basic authentication is chosen, you have to use encryption at a lower layer—for example, by using HTTPS instead of HTTP.

11.1.6 Enabling remoting in the enterprise

Remote administration is most likely to be performed against the servers in your environment. As you've seen, the newer versions of Windows Server have PowerShell remoting enabled by default. If you have older servers, you don't want to have to enable remoting on them individually because you may be dealing with tens, hundreds, or thousands of machines. Obviously, you can't use PowerShell remoting to turn on remoting, so you need another way to push configuration out to a collection of machines. This is exactly what Group Policy is designed for. You can use Group Policy to enable and configure remoting as part of the machine policy that gets pushed out.

PowerShell depends on the WinRM (Windows Remote Management) service for its operation. Your Group Policy needs to:

- Ensure the WinRM service will start automatically and is started.
- Configure WinRM to accept remoting requests.
- Configure Windows Firewall to allow remoting requests.

Instructions on creating a suitable Group Policy are available at http://mng.bz/3aHW.

11.2 Applying PowerShell remoting

With remoting services enabled, you can start to use them to get your work done. In this section, we're going to look at ways you can apply remoting to solve management problems. We'll start with some simple remoting examples. Next, we'll work with more complex examples where we introduce concurrent operations. Then you'll apply the principles you've learned to solve a specific problem: how to implement a

multi-machine configuration monitor. You'll work through this problem in a series of steps, adding more capabilities to your solution, resulting in a simple but fairly complete configuration monitor. Let's start with the most basic examples.

11.2.1 Basic remoting examples

Building on our "Hello world" example from chapter 1, the most basic example of remoting is

```
Invoke-Command -ComputerName Servername -ScriptBlock {'Hello world'}
```

The first thing to notice is that `Invoke-Command` takes a scriptblock to specify the actions. This pattern should be familiar by now—you've seen it with `ForEach-Object` and `Where-Object` many times. The `Invoke-Command` does operate a bit differently, though. It's designed to make remote execution as transparent as possible. For example, if you want to sort objects, the local command looks like this:

```
PS> 1..3 | sort -Descending
```

Now if you want to do the sorting on the remote machine, you'd do this:

```
PS> 1..3 |
Invoke-Command -ComputerName localhost -ScriptBlock {sort -Descending}
```

You're splitting the pipeline across local and remote parts, and the scriptblock is used to demarcate which part of the pipeline should be executed remotely.

> **NOTE** Localhost is used to set a remote session to your local machine for testing purposes. You could use the machine name if preferred or `$ENV:COMPUTERNAME`.

This works the other way as well:

```
PS> Invoke-Command -ComputerName localhost -ScriptBlock { 1..3 } |
sort -Descending
```

Here you're generating the numbers on the remote computer and sorting them locally. Scriptblocks can contain more than one statement. This implies that the semantics need to change a bit. Whereas in the simple pipeline case streaming input into the remote command was transparent, when the remote command contains more than one statement, you have to be explicit and use the `$input` variable to indicate where you want the input to go. That looks like the following:

```
PS> 1..3 | Invoke-Command -ComputerName localhost -ScriptBlock {
  'First'
  $input | sort -Descending
  'Last'
}
First
3
```

```
2
1
Last
```

The scriptblock argument to `Invoke-Command` in this case contains three statements. The first emits the string `'First'`, the second does the sort on the input, and the third emits the string `'Last'`.

What happens if you don't specify input? Nothing is emitted between `'First'` and `'Last'`. Because `$input` wasn't specified, the input objects were never processed. You'll need to keep this in mind when you start to build a monitoring solution.

Now let's look at how concurrency—multiple operations occurring at the same time—impacts your scripts.

11.2.2 Adding concurrency to the examples

In chapter 1, we talked about how each object passed completely through all states of a pipeline, one by one. This behavior changes with remoting because the local and remote commands run in separate processes that are executing concurrently. This means you now have two threads of execution—local and remote—and that can have an effect on the order in which things are executed. Consider the following statement:

```
PS> 1..3 | foreach { Write-Host $_ -ForegroundColor green;
$_; Start-Sleep 5 } | Write-Host

1
1
2
2
3
3
```

This statement sends a series of numbers down the pipeline. In the body of the `foreach` scriptblock, the value of the current pipeline object is written to the screen (in green) and then passed to the next state in the pipeline. This last stage also writes the object to the screen (in standard color). Given that you know each object is processed completely by all stages of the pipeline, the order of the output is as expected. The first number is passed to `foreach`, where it's displayed and then passed to `Write-Output`, where it's displayed again, so you see the sequence 1, 1, 2, 2, 3, 3.

> **NOTE** `Start-Sleep` is used to build sufficient pauses into the execution so that you can see what's happening. Run the code without `Start-Sleep` to see the difference.

Now let's run this command again using `Invoke-Command` in the final stage:

```
PS> 1..3 | foreach {
  Write-Host -ForegroundColor green  $_
  $_; Start-Sleep 5 } |
    Invoke-Command -ComputerName localhost -ScriptBlock { Write-Host }
```

```
1
2
1
3
2
3
```

The order has changed—you see 1 and 2 from the local process in green on a color display, then you see 1 from the remote process (in your normal foreground text color), and so on. The local and remote pipelines are executing at the same time, which is what's causing the changes to the ordering. Predicting the order of the output is made more complicated by the use of buffering and timeouts in the remoting protocol.

You used the `Start-Sleep` command in these examples to force these visible differences. If you take out this command, you'll get a different pattern:

```
PS> 1..3 | foreach { Write-Host $_ -ForegroundColor green ; $_ } |
 Invoke-Command -ComputerName localhost -ScriptBlock { Write-Host }

1
2
3
1
2
3
```

This time, all the local objects are displayed (in green) and then passed to the remoting layer, where they're buffered until they can be delivered to the remote connection. This way, the local side can process all objects before the remote side starts to operate. Concurrent operation and buffering make it appear a bit unpredictable, but if you didn't have the `Write-Host`s in place, it would be unnoticeable. The important thing to understand is that objects being sent to the remote end will be processed concurrently with the local execution. That means the remoting infrastructure doesn't have to buffer everything sent from the local end before starting execution.

Up to now, you've been passing only simple commands to the remote end. But because `Invoke-Command` takes a scriptblock, you can, in practice, send pretty much any valid PowerShell script. You'll take advantage of this fact in the next section when you start to build your multi-machine monitor.

> **NOTE** Why does remoting require scriptblocks? Two reasons: Scriptblocks are always compiled locally so you'll catch syntax errors as soon as the script is loaded, and using scriptblocks limits vulnerability to code injection attacks by validating the script before sending it.

11.2.3 Solving a real problem: multi-machine monitoring

In this section, you're going to build a solution for a real management problem: multi-machine monitoring. With this solution, you're going to gather some basic health

information from the remote host. The goal is to use this information to determine when a server may have problems such as out of memory, out of disk, or reduced performance due to a high faulting rate. You'll gather the data on the remote host and return it as a hashtable so you can look at it locally.

Your requirements are as follows:

- Collect the amount of free space on the C: drive from the `Get-PSDrive` command.
- Collect the page fault rate retrieved using CIM (WMI).
- Collect the processes consuming the most CPU from `Get-Process` with a pipeline.
- Collect the processes that have the largest working set, also from `Get-Process`.
- Ensure the list of computers you monitor aren't hardcoded into the script; the computers to monitor will be listed in a file.
- Monitor each computer on specific days with the results stored in the file.
- Apply a throttle limit to control how many simultaneous machines are monitored.
- Parameterize the script for ease of use.

This listing shows a solution to the problem using the techniques you've learned so far in the book.

Listing 11.1 Parameterized monitoring script

```
param (
  [string] $serverFile = 'servers.txt',
  [int] $throttleLimit = 10,
  [int] $numProcesses = 5
)

$gatherInformation ={
    param ([int] $procLimit = 5)
    @{
        Date = Get-Date
        FreeSpace = (Get-PSDrive c).Free
        PageFaults = (Get-WmiObject `
            Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
        TopCPU = Get-Process |
                 Sort-Object CPU -Descending |
                 Select-Object -First $procLimit
        TopWS =  Get-Process |
                 Sort-Object WS -Descending |
                 Select-Object -First $procLimit
    }
}

$servers = Import-CSV $serverfile |
    Where-Object { $_.Day -eq (Get-Date).DayOfWeek } |
    foreach { $_.Name }

Invoke-Command -ThrottleLimit $throttleLimit -ComputerName $servers `
       -ScriptBlock $gatherInformation `
       -ArgumentList $numProcesses
```

1 Define parameters

2 Create scriptblock

3 Get servers to monitor

4 Perform monitoring

The first two parameters ❶ are obvious: `$ServerFile` is the name of the file containing the list of servers to check, and `$throttleLimit` is the throttle limit (number of simultaneous connections the monitoring script makes to remote machines). The default throttle limit for `Invoke-Command` is 32. We're deliberately lowering that to ensure we don't overload the local machine.

The third parameter, `$numProcesses`, controls the number of process objects to include in the `TopCPU` and `TopWS` entries in the table returned from the remote host. Although you could in theory trim the list that gets returned locally, you can't add to it, so you need to evaluate this parameter on the remote end to get full control. That means it has to be a parameter to the remote command. This is another reason scriptblocks are useful. You can add parameters to the scriptblock that's executed on the remote end.

The scriptblock to be passed to the remote machines is defined ❷. Notice the parameter on the scriptblock that's executed on the remote end. That's how the number of processes to return is passed to the remote server.

The list of servers is derived from the input file ❸. The contents of servers.txt would look something like this:

```
Name,Day
W16DSC01,Monday
W16TGT01,Tuesday
W16PWA01,Wednesday
W16DSC02,Saturday
W16CN01,Thursday
W16AS01,Friday
```

When you load the servers, you'll do some processing on this list to determine the current day of the week and decide which servers need monitoring.

The final step ❹ is to use `Invoke-Command` to send the scriptblock to the appropriate servers. Figure 11.4 shows the script in action.

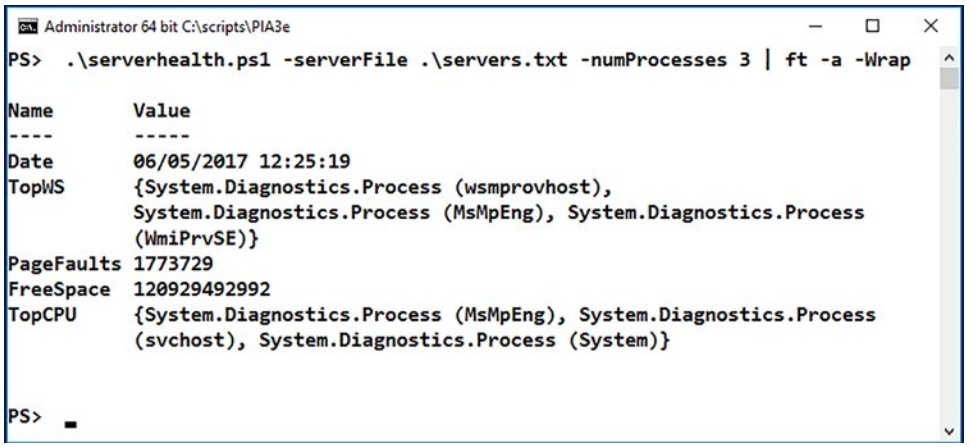


Figure 11.4 Listing 11.1 in action

Listing 11.1 was saved as serverhealth.ps1. We decided we needed only the top three processes rather than the default five. The data is returned as a hashtable. Notice that the process data is embedded as objects. You'd need to perform further processing locally if you wanted to drill down into the process objects.

The result is that, with a small amount of code, you've created a flexible framework for an agentless distributed health monitoring system. With this system, you can run this health model on *any* machine without having to worry about whether the script is installed on that machine or whether the machine has the correct version of the script. It's always available and always the right version because the infrastructure is pushing it out to the target machines. You can even have different files of server names if required.

> **NOTE** What we're doing here isn't what most people would call monitoring, which usually implies a continual semi-real-time mechanism for noticing a problem and then generating an alert. This system is certainly not real time, and it's a pull model, not a push. This solution is more appropriate for configuration analysis.

You now have an idea of how to use remoting to execute a command on a remote server. This is a powerful mechanism, but sometimes you need to send more than one command to a server; for example, you might want to run multiple data-gathering scripts, one after the other, on the same machine. Because there's a significant overhead in setting up each remote connection, you don't want to create a new connection for every script you execute. Instead, you want to be able to establish a persistent connection to a machine, run all the scripts, and then shut down the connection.

11.3 PowerShell remoting sessions and persistent connections

In the previous section, you learned how to run individual scriptblocks on remote machines. From the user's point of view, the `Invoke-Command` operation is simple, but under the covers the system has to do a lot of work creating, using, and deleting the connection, which makes creating a new connection each time a costly proposition. Also, you can't maintain any state—things like variable settings or function definitions—on the remote host.

To address these issues, in this section we'll show you how to create persistent connections called *sessions* that will give you much better performance when you want to perform a series of interactions with the remote host as well as allow you to maintain remote state. In the simplest terms, a *session* is the environment where PowerShell commands are executed. This is true even when you run the console host, PowerShell.exe. The console host program creates a local session that it uses to execute the commands you type. This session remains alive until you exit the program. When you use remoting to connect to another computer, you're also creating one remote session for every local session you remote from until explicitly closed. An instance of `wsmprovhost.exe` per connecting session will run on the remote host as long as that session is open.

Each session contains all the things you work with in PowerShell—all the variables, all the functions that are defined, and the history of the commands you typed—and each session is independent of any other session. If you want to work with these sessions, you need a way to manipulate them. You do this in the usual way: through objects and cmdlets. PowerShell represents sessions as objects that are of type `PSSession`.

By default, every time you connect to a remote computer by name with `Invoke-Command`, a new `PSSession` object is created to represent the connection to that remote machine. If you're going to run more than one command on a computer, you need a way to create persistent connections to that computer. You can do this with `New-PSSession`; the syntax for this cmdlet is shown in figure 11.5.

```
New-PSSession [[-ComputerName] <string[]>] [-Credential <pscredential>]
[-Name <string[]>] [-EnableNetworkAccess] [-Port <int>] [-UseSSL]
[-ConfigurationName <string>] [-ApplicationName <string>]
[-ThrottleLimit <int>] [-SessionOption <PSSessionOption>]
[-Authentication <AuthenticationMechanism>]
[-CertificateThumbprint <string>] [<CommonParameters>]
```

Figure 11.5 The syntax for the `New-PSSession` cmdlet. This cmdlet is used to create persistent connections to a remote computer.

This command has many of the same parameters that you saw in `Invoke-Command`. The difference is that, for `New-PSSession`, these parameters are used to configure the persistent session instead of the transient sessions you saw being created by `Invoke-Command`. The `PSSession` object returned from `New-PSSession` can then be used to specify the destination for the remote command instead of the computer name.

The lifetime of the session begins with the call to `New-PSSession` and persists until it's explicitly destroyed by the call to `Remove-PSSession`. Let's look at an example that illustrates how much of a performance difference sessions can make. You'll run `Get-Date` five times using `Invoke-Command` and see how long it takes using `Measure-Command` (which measures command execution time).

First, execute the test without sessions:

```
PS> Measure-Command { 1..5 |
foreach { Invoke-Command W16TGT01 {Get-Date} } } |
Format-Table -AutoSize TotalSeconds

TotalSeconds
------------
   4.7129865
```

The result from `Measure-Command` shows that each operation appears to be taking a little under one second. Modify the example to create a session at the beginning and then reuse it in each call to `Invoke-Command`:

```
PS> Measure-Command {
  $s = New-PSSession W16TGT01
  1..5 |
```

```
  foreach { Invoke-Command $s {Get-Date} }
   Remove-PSSession $s
} |
Format-Table -AutoSize TotalSeconds

TotalSeconds
------------
   0.8096949
```

This output shows that it's taking about one-sixth the time as the first command. Increasing the number of remote invocations from 5 to 50 results in an execution time of 1.4997587 seconds. Clearly, for this simple example, the time to set up and break down the connection totally dominates the execution time. Other factors affect real scenarios, such as network performance, the size of the script, and the amount of information being transmitted. Still, it's obvious that when multiple interactions are required, using a session will result in substantially better performance.

The downside is that persistent sessions will monopolize your machine's limited resources, so if you forget to close a session, you may soon hit the limits set (max user connections, max connections per server). Cleaning up unrequired sessions is definitely in your best interest. The two most expensive penalties with remoting are setting up the session and serializing the return data. Filtering on the remote machine to reduce the amount of data to be returned can also significantly improve performance.

11.3.1 Additional session attributes

This section describes some `PSSession` attributes that can have an impact on the way you write your scripts.

Sessions and hosts

The host application running your scripts can impact the portability of your scripts if you become dependent on specific features of that host. (This is why PowerShell module manifests include the `PowerShellHostName` and `PowerShellHostVersion` elements.) Dependency on specific host functionality is a consideration with remote execution because the remote host implementation is used instead of the normal interactive host. This is necessary to manage the extra characteristics of the remote or job environments. This host shows up as a process named `wsmprovhost` corresponding to the executable `wsmprovhost.exe`. This host supports only a subset of the features available in the normal interactive PowerShell hosts.

Session isolation

Another point is the fact that each session is configured independently when it's created, and once it's constructed, it has its own copy of the engine properties, execution policy, function definitions, and so on. This independent session environment exists for the duration of the session and isn't affected by changes made in other sessions. This principle is called *isolation*—each session is isolated from, and therefore not affected by, any other session.

ONLY ONE COMMAND RUNS AT A TIME

A final characteristic of a session instance is that you can run only one command (or command pipeline) in a session at one time. If you try to run more than one command at a time, a "session busy" error will be raised. But there's some limited command queuing: if there's a request to run a second command synchronously (one at a time), the command will wait up to four minutes for the first command to be completed before generating the "session busy" error. But if a second command is requested to run asynchronously—without waiting—the busy error will be generated immediately.

With some knowledge of the characteristics and limitations of PowerShell sessions, you can start to look at how to use them.

11.3.2 Using the New-PSSession cmdlet

In this section, you'll learn how to use the `New-PSSession` cmdlet. Let's start with an example. First, you'll create a `PSSession` on the local machine by specifying `localhost` as the target computer:

```
PS> $s = New-PSSession -ComputerName localhost
```

> **NOTE** By default a user must be running with elevated privileges to create a session on the local machine. You'll see how to change the default setting later.

You now have a `PSSession` object in the `$s` variable that you can use to execute remote commands. Earlier we said each session runs in its own process. You can confirm this by using the `$PID` session variable to see what the process ID of the session process is. First, run this code in the remote session

```
PS> Invoke-Command -Session $s -ScriptBlock {$PID}
9436
```

and you see that the process ID is 9436. When you get the value in the local session by typing `$PID` at the command line, as shown here

```
PS> $PID
8528
```

you see that the local process ID is 8528.

> **NOTE** The numbers you see may well be different than those shown here. The important point is that the `$PID` values are different when running locally and through a remoting session.

Now define a variable in the remote session:

```
PS> Invoke-Command -Session $s -ScriptBlock {$x=1234}
```

With this command, you've set the variable `$x` in the remote session to 1234. This works in much the same way as it does in the local case—changes to the remote environment

are persisted across the invocations. You can define a function and make it reference the `$x` variable you defined earlier:

```
PS> Invoke-Command -Session $s -ScriptBlock {
  function hi {"Hello there, x is $x"}
}
PS> Invoke-Command -Session $s -ScriptBlock {hi}
Hello there, x is 1234
```

You get the preserved value.

> **NOTE** We've had people ask whether other users on the computer can see the sessions we're creating. As mentioned earlier, this isn't the case. Users have access only to the remote sessions they create and only from the sessions they were created from. There's no way for one session to connect to another session that it didn't itself create. The only aspect of a session that may be visible to another user is the existence of the `wsmprovhost` process hosting the session.

As you've seen, remote execution is like the local case . . . well, almost. You have to type `Invoke-Command` every time. If you're executing a lot of interactive commands on a specific machine, this task quickly becomes annoying. PowerShell provides a much better way to accomplish this type of task, as you'll see in the next section.

11.3.3 Interactive sessions

In the previous sections, you learned how to issue commands to remote machines using `Invoke-Command`. This approach is effective but gets annoying for more interactive types of work. To make this scenario easier, you can start an *interactive session* using the `Enter-PSSession` cmdlet. Once you're in an interactive session, the commands you type are automatically passed to the remote computer and executed without having to use `Invoke-Command`. Let's try this out. You'll reuse the session you created in the previous section. In that session, you defined the variable `$x` and the function `hi`. To enter interactive mode during this session, you'll call `Enter-PSSession`, passing in the session object, as shown in figure 11.6.

> **NOTE** Only interactive commands are transmitted when you use `Enter-PSSession`. You can't use it in a script and pass commands to the session.

As soon as you enter interactive mode, you see that the prompt changes: it now displays the name of the machine you're connected to and the current directory.

> **NOTE** The default prompt can be changed in the remote session in the same way it can be changed in the local session. If you have a prompt definition in your profile, you may be wondering why that wasn't used. We'll get to that later when we look at some of the things you need to keep in mind when using remoting.

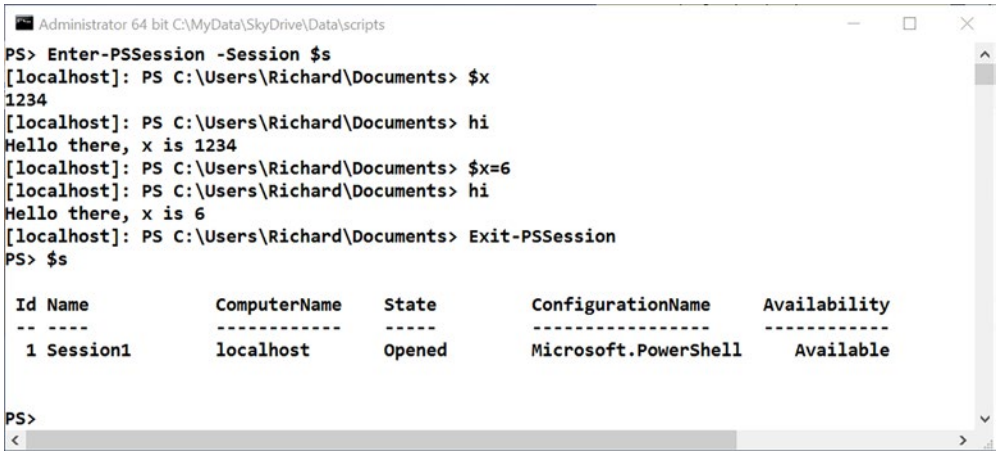


Figure 11.6 Using a `PSSession` for interactive remoting

You can see from the code being run in the figure that the value of `$x` is preserved (1234) and the `hi` function you defined is also available. Changing the value of `$x` and then rerunning the `hi` function shows the new value displayed in the output.

You can exit an interactive remote session either by using the `exit` keyword or by using the `Exit-PSSession` cmdlet. You see that the prompt changed back and the session still exists. It will persist until explicitly removed with `Remove-PSSession` or the PowerShell instance is closed. You can enter and exit a session as often as you need to as long as it's not removed in the interim.

Another useful feature to consider is the fact that you can have more than one session open at a time. This means you can pop back and forth between multiple computers as needed, which makes dealing with multiple machines convenient.

More differences exist between the pattern where you used `Invoke-Command` for each command and the interactive mode. In the non-interactive `Invoke-Command` case, the remote commands send objects back, where they're formatted on the local machine. In the interactive remoting case, the objects are formatted on the *remote* machine, and simple strings are sent to the local machine to be displayed. Usually this won't matter, but cultural information such as dates and object formatting may be impacted.

Finally, as with the non-interactive remoting case, you can run an interactive session in a temporary session by passing the name of the computer instead of an existing `PSSession`. Using the `PSSession` has the advantage that you can enter and exit the remote session and have the remote state preserved between activities. If the name of the computer is passed in, the connection will be torn down when you exit the session. Because a remote session involves creating a remote host process, forgetting to close your sessions can waste resources. At any point, you can use `Get-PSSession` to get a list of the open sessions you currently have and use `Remove-PSSession` to close them as appropriate.

By now, you should be comfortable with creating and using persistent remote sessions. What we haven't spent much time on yet is how to manage all these connections you're creating.

11.3.4 Managing PowerShell sessions

Each `PSSession` is associated with an underlying `Windows` process. As such, it consumes significant resources even when no commands are being executed in it. You should delete `PSSessions` that are no longer needed. This reduces the memory usage and similar drains on the remote system. At the same time, creating new `PSSessions` also puts a load on the system, consuming additional CPU resources to create each new process. When managing your resource consumption, you need to balance the cost of creating new sessions against the overhead of maintaining multiple sessions. There's no hard-and-fast rule for deciding what this balance should be. In the end, you should decide on an application-by-application basis.

To get a list of the existing `PSSessions`, you use the `Get-PSSession` command, and to remove sessions that are no longer needed, you use the `Remove-PSSession` cmdlet. The `Remove-PSSession` cmdlet closes the `PSSession`, which causes the remote process to exit and frees up all the resources it held. Removing the session also frees up local resources like the network connection used to connect to the remote session.

With PowerShell v2 you can view the sessions on the local machine, whereas PowerShell v3 and later enable you to see the sessions on remote as well as local machines. On a local machine, you'll see something like this:

```
PS>  Get-PSSession |
Format-List Id, Name, ComputerName, ComputerType, State,
ConfigurationName, Availability

Id                : 1
Name              : Session1
ComputerName      : W16TGT01
ComputerType      : RemoteMachine
State             : Opened
ConfigurationName : Microsoft.PowerShell
Availability      : Available
```

The remote machine (use the `-ComputerName` parameter) may give you results like this:

```
PS>  Get-PSSession -ComputerName W16TGT01 |
Format-List Id, Name, ComputerName, ComputerType, State,
ConfigurationName, Availability

Id                : 1
Name              : Session1
ComputerName      : W16TGT01
ComputerType      : RemoteMachine
State             : Opened
ConfigurationName : Microsoft.PowerShell
Availability      : Available
```

```
Id                : 3
Name              : Session1
ComputerName      : W16TGT01
ComputerType      : RemoteMachine
State             : Disconnected
ConfigurationName : Microsoft.PowerShell
Availability      : Busy
```

In this case, the session with an `Id` of `1` (state is `Opened`) is the session created from your local machine. The session with an `Id` of `3` is another session to the remote machine—in this case, created from a third machine. We know this because we created them. Unfortunately, there's no way to tell who created a session connected to a remote machine or from which machine it was created. Notice that session `Id 3` is shown with a state of `Disconnected`. This means you aren't connected to it.

> **TIP** The ID number will change every time you access the sessions on the remote machine created by a PowerShell session other than your own. It's worth giving your session distinctive names so that you can easily distinguish between sessions.

On the client end, if you don't explicitly remove the sessions or set timeouts, local sessions will remain open until you end your PowerShell session. But what happens if the client fails for some reason without closing its sessions? If the PowerShell session is closed or the local machine crashes, the remote session will be terminated. If network connectivity is lost or the session times out (the default is two hours), the session may be put into a disconnected state. You can also put a session into a disconnected state manually.

> **NOTE** Commands continue to run in a disconnected session. You can even deliberately create a disconnected session using the `-InDisconnectedSession` parameter of `Invoke-Command`.

The sessions shown earlier in this section have been re-created with distinctive names:

```
PS>  Get-PSSession -ComputerName W16TGT01 |
Format-Table Id, Name, ComputerName, State,
Availability -AutoSize

Id Name         ComputerName         State  Availability
-- ----         ------------         -----  ------------
 4 FromW16AS01  W16TGT01            Opened     Available
 5 FromW16DSC01 W16TGT01      Disconnected          Busy
```

`FromW16AS01` is the one from our local machine. That session can be disconnected:

```
PS> Disconnect-PSSession -Name FromW16AS01

Id Name         ComputerName          State  Availability
-- ----         ------------          -----  ------------
 4 FromW16AS01  W16TGT01       Disconnected          None
```

Notice that state changes to `Disconnected` and availability changes to `None`. After closing the PowerShell session that created the session `FromW16AS01` and opening a new PowerShell session, using `Get-PSSession` to test for a session will return nothing as expected—we haven't created any remoting sessions in that PowerShell session.

Now try getting the sessions on the remote server we were working with:

```
PS>  Get-PSSession -ComputerName W16TGT01 |
Format-Table Id, Name, ComputerName, State,
Availability -AutoSize

Id Name         ComputerName        State Availability
-- ----         ------------        ----- ------------
 1 FromW16AS01  W16TGT01     Disconnected         None
 2 FromW16DSC01 W16TGT01     Disconnected         Busy
```

You can reconnect to the session—in this case session `FromW16AS01`:

```
PS> Connect-PSSession -ComputerName W16TGT01
Connect-PSSession : Cannot connect PSSession "FromW16DSC01",
either because it is not in the Disconnected state, or it
is not available for connection.
At line:1 char:1
+ Connect-PSSession -ComputerName W16TGT01
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: ([PSSession]
  W16TGT01:PSSession) [Connect-PSSession],
RuntimeExcept   ion
    + FullyQualifiedErrorId : PSSessionConnectFailed,Microsoft.PowerShell.
  Commands.ConnectPSSessionCommand

Id Name        ComputerName  ComputerType   State  Availability
-- ----        ------------  ------------   -----  ------------
 3 FromW16AS01 W16TGT01      RemoteMachine  Opened Available
```

You can connect to the session `FromW16AS01`, but you can't connect to the session from the third machine because it already has an open connection (hold that thought). Once connected, your session is available for use again:

```
PS> $s = Get-PSSession -Name FromW16AS01
PS> Invoke-Command -Session $s -ScriptBlock `
{Get-CimInstance Win32_OperatingSystem}

SystemDirectory     BuildNumber Version    PSComputerName
---------------     ----------- -------    --------------
C:\Windows\system32 14393       10.0.14393 W16TGT01
<output truncated for brevity>
```

If a session is disconnected from its original host, you can connect to it from either the original host or another machine. After disconnecting the session `FromW16DSC01` from its original host and testing available sessions on the local machine,

```
PS>  Get-PSSession -ComputerName W16TGT01 |
Format-Table Id, Name, ComputerName, State,
Availability -AutoSize
```

```
Id Name          ComputerName        State Availability
-- ----          ------------        ----- ------------
 3 FromW16AS01   W16TGT01           Opened    Available
 6 FromW16DSC01 W16TGT01     Disconnected         None
```

you can see that the session `FromW16DSC01` is disconnected and availability is shown as `None`. Connect to it in a similar way as before:

```
PS> Connect-PSSession -Name FromW16DSC01 -ComputerName W16TGT01
PS>  Get-PSSession -ComputerName W16TGT01 |
Format-Table Id, Name, ComputerName, State,
Availability -AutoSize
```

```
Id Name          ComputerName  State Availability
-- ----          ------------  ----- ------------
 3 FromW16AS01   W16TGT01     Opened    Available
 7 FromW16DSC01 W16TGT01     Opened    Available
```

Disconnected sessions created by you on the local or other machine can be reconnected and used as shown. You can even connect to disconnected sessions created by other people as long as you have the credential details they used to create the session originally.

You can also use a PowerShell remoting session for copying files to and from a remote machine.

11.3.5 Copying files across a PowerShell remoting session

PowerShell remoting is used to run commands on remote machines, as you saw in earlier sections, and have the results returned to you. In PowerShell v2–v4 you couldn't copy files using a PowerShell remoting session. This changed in PowerShell v5 with the introduction of the `-FromSession` and `-ToSession` parameters on the `Copy-Item` cmdlet. Both of these new parameters take a *single* `PSSession` object as input.

This concept is best described by an example. Start by creating remoting sessions to two machines:

```
PS> $s1 = New-PSSession -ComputerName W16TGT01
PS> $s2 = New-PSSession -ComputerName W16DSC02
```

Now create a file on a remote machine:

```
PS> Invoke-Command -Session $s1 -ScriptBlock {
Get-Process | Out-File -FilePath c:\scripts\proc.txt}
```

You can copy the file from the remote machine to the local machine:

```
PS> Copy-Item -Path c:\scripts\proc.txt -FromSession $s1
```

Check that it arrived and then copy it to the second machine:

```
PS> Copy-Item -Path proc.txt  -Destination C:\Scripts\ -ToSession $s2
```

A simple check confirms that the copy occurred:

```
PS> Invoke-Command -Session $s2 `
-ScriptBlock {Get-ChildItem -Path C:\Scripts\}
```

Everyone looks at the sequence of commands and thinks we can combine the copy steps:

```
PS> Copy-Item -Path c:\scripts\proc.txt  -Destination C:\Scripts\ `
-FromSession $s1  -ToSession $s2
Copy-Item : '-FromSession' and '-ToSession' are mutually exclusive and cannot
  be specified at the same time.
At line:1 char:1
+ Copy-Item -Path c:\scripts\proc.txt  -Destination C:\Scripts\ -FromSe ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (Microsoft.Power...namicParame
  ters:CopyItemDynamicParameters) [Copy-Item],
   ArgumentException
    + FullyQualifiedErrorId : InvalidInput,Microsoft.PowerShell.Commands.
  CopyItemCommand
```

Unfortunately, we can't. The `-FromSession` and `-ToSession` parameters are mutually exclusive.

> **NOTE** This isn't obvious from the help file because the parameters are shown in the same parameter set and their mutual exclusivity isn't mentioned in the text.

You can copy multiple files across a PowerShell remoting session using wildcards to define the files.

11.4 Implicit remoting

When doing non-interactive remoting, you have to call `Invoke-Command` every time you want to execute a remote operation. You can avoid this task by using `Enter-PSSession` to set up a remote interactive session. This approach makes remote execution easy but at the cost of making local operations difficult. In this section, we'll look at a mechanism that makes both local and remote command execution easy. This mechanism is called *implicit remoting*.

> **NOTE** For implicit remoting to work, the execution policy on the client machine has to be configured to allow scripts to run, typically by setting it to `RemoteSigned`. This is necessary because implicit remoting generates a temporary module, and PowerShell must be allowed to execute scripts in order to load this module. If execution policy is set to `Restricted` or `AllSigned`, it won't be able to do this. This requirement applies only to the local client machine. A remote server can still use a more restrictive policy. See section 7.1.1 for more information about execution policy.

The goals of implicit remoting are to make the fact that remote operations are occurring invisible to the user and to have all operations look as much like local operations as possible. You can accomplish this goal by generating *local proxy functions* that run the

remote commands under the covers. The user calls the local proxy, which takes care of the details involved in making the remote command invocation.

The net effect is that everything looks like a local operation because everything *is* a local operation.

11.4.1 Using implicit remoting

To set up the remote proxy functions mentioned in the previous section, use the `Import-PSSession` cmdlet. The syntax for this cmdlet is shown in figure 11.7.

```
Import-PSSession [-Session] <PSSession> [[-CommandName] <string[]>]
[[-FormatTypeName] <string[]>] [-Prefix <string>] [-DisableNameChecking]
[-AllowClobber] [-ArgumentList <Object[]>]
[-CommandType <CommandTypes>] [-Module <string[]>]
[-FullyQualifiedModule <ModuleSpecification[]>]
[-Certificate <X509Certificate2>] [<CommonParameters>]
```

Figure 11.7 The syntax for the `Import-PSSession` cmdlet. This cmdlet is used to create local proxy commands that invoke the corresponding remote command on the target computer.

Let's explore how this cmdlet works by walking through an example. You'll create a `PSSession` and then define a function in that session. The goal is to be able to execute this remote function as though it were defined locally. You want to implicitly remote the function. To do that, you call `Import-PSSession`, which generates a function that you can call locally. This local function does the remote call on your behalf—it acts as your *proxy*.

You'll begin by creating the connection to a remote machine. You may need to get credentials for the remote host.

> **NOTE** In a domain environment, this step is unnecessary as long as your user account has sufficient privileges to access the remote endpoint. But if you want to log on as a different user, credentials will be required.

Establish a session on the remote machine, using credentials if necessary, as shown in figure 11.8.

Next, you'll use `Invoke-Command` to define a new function on the remote machine. This is the command you'll import:

```
PS> Invoke-Command -Session $s -ScriptBlock {
function Get-Bios {Get-WmiObject Win32_Bios}}
```

The new remote function, called `Get-Bios`, uses Windows Management Instrumentation (WMI) to retrieve information about the BIOS on the remote machine. Invoke this function through explicit remoting using `Invoke-Command` so you can see that it

Figure 11.8 Example of implicit remoting

returns a set of information about the BIOS on the remote machine. Now use `Import-PSSession` to create a local proxy for this command:

```
PS> Import-PSSession -Session $s -CommandName Get-Bios

ModuleType Version    Name                ExportedCommands
---------- -------    ----                ----------------
Script     1.0        tmp_4qxsxsjw.5m2    Get-Bios
```

You might recognize the output from this command—it's the same thing you see when you do `Get-Module`. You now have a local `Get-Bios` command. Try running it:

```
PS> Get-Bios

SMBIOSBIOSVersion : Hyper-V UEFI Release v1.0
Manufacturer      : Microsoft Corporation
Name              : Hyper-V UEFI Release v1.0
SerialNumber      : 8265-3792-6973-7306-2850-7895-37
Version           : VRTUAL - 1
```

You get the same result you saw when you did the explicit remote invocation but without having to do any extra work to access the remote machine. The proxy command

did that for you. This is the goal of implicit remoting: to make the fact that the command is being executed remotely invisible.

> **NOTE** This is a useful technique because you need to import the Exchange management module into your session if you're administering an Exchange server over a PowerShell remoting session.

Let's see how it all works.

11.4.2 How implicit remoting works

When the user requests that a command be imported, a message is sent to the remote computer for processing. The import request processor looks up the command and retrieves the metadata (the `CommandInfo` object) for that command. That metadata is processed to simplify it, removing things like complex type attributes. Only the core remoting types are passed along. This metadata is received by the local machine's proxy function generator. It uses this metadata to generate a function that will implicitly call the remote command.

Let's take a closer look at what the generated proxy looks like. You can see the imported `Get-Bios` command using `Get-Command`:

```
PS> Get-Command Get-Bios

CommandType Name     Version Source
----------- ----     ------- ------
Function    Get-Bios 1.0     tmp_4qxsxsjw.5m2
```

The output shows that you have a local function called `Get-Bios`. You can look at the definition of that function by using the `Definition` property on the `CommandInfo` object returned by `Get-Command`.

Listing 11.2 Definition of the `Get-Bios` proxy function

```
param(
    [switch]${AsJob}
)

Begin {
        try {
            $positionalArguments =
            & $script:NewObject collections.arraylist
            foreach ($parameterName in
                  $PSBoundParameters.BoundPositionally)
            {
                $null = $positionalArguments.Add(
                $PSBoundParameters[$parameterName] )
                $null = $PSBoundParameters.Remove($parameterName)
            }
            $positionalArguments.AddRange($args)
```

```
            $clientSideParameters =
            Get-PSImplicitRemotingClientSideParameters`
            $PSBoundParameters $False

            $scriptCmd = { & $script:InvokeCommand `
                           @clientSideParameters `
                           -HideComputerName `
                           -Session (Get-PSImplicitRemotingSession `
                           -CommandName 'Get-Bios') `
                           -Arg ('Get-Bios', $PSBoundParameters,
                            $positionalArguments) `
                           -Script { param($name, $boundParams,
                            $unboundParams) & $name @boundParams
                            @unboundParams }`
                         }

            $steppablePipeline =
            $scriptCmd.GetSteppablePipeline($myInvocation.CommandOrigin)
            $steppablePipeline.Begin($myInvocation.ExpectingInput,
            $ExecutionContext)
        } catch {
            throw
        }
    }
    Process {
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}
    End {
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}

    # .ForwardHelpTargetName Get-Bios
    # .ForwardHelpCategory Function
    # .RemoteHelpRunspace PSSession
```

Even though this output has been reformatted a bit to make it more readable, it's a pretty complex function and uses many of the more sophisticated features covered in previous chapters. It uses advanced functions, splatting, scriptblocks, and steppable pipelines. Fortunately, you never have to write these functions yourself.

> **NOTE** You don't have to create proxy functions for this particular scenario, but in section 11.5.2 you saw how this technique can be powerful in extending the PowerShell environment.

The `Import-PSSession` cmdlet does this for you. It will create a proxy function for each command it's importing, which could lead to many commands. As well as

generating proxy functions on your behalf, `Import-PSSession` creates a module to contain these functions.

The module name and path are temporary generated names. This module also defines an `OnRemove` handler (see chapter 9) to clean up when the module is removed. To see the contents of the module, you can look at the temporary file that was created by using the module's `Path` property:

```
PS> Get-Content (Get-Command Get-Bios).Module.Path
```

Alternatively, you can save the session to an explicitly named module for reuse with `Export-PSSession`. You'll save this session as a module called bios:

```
PS> Export-PSSession -OutputModule bios -Session $s `
-type function -CommandName Get-Bios -AllowClobber

        Directory: C:\Users\Richard\Documents\WindowsPowerShell\Modules\bios

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        08/05/2017     11:51             99 bios.format.ps1xml
-a----        08/05/2017     11:51            528 bios.psd1
-a----        08/05/2017     11:51          11627 bios.psm1
```

Executing this command creates a new module in your user module directory. It creates the script module file (.psm1), the module manifest (.psd1), and a file containing formatting information for the command. You use the `-AllowClobber` parameter because the export is using the remote session to gather the data. If it finds a command being exported that already exists in the caller's environment, that would be an error. Because `Get-Bios` already exists, you have to use `-AllowClobber`.

Import the module into a new PowerShell session—remember to open it with elevated privileges:

```
PS> Import-Module bios
```

It returns right away. It can do this because it hasn't set up the remote connection yet. This will happen the first time you access one of the functions in the module. Run `Get-Bios`:

```
PS> Get-Bios
Creating a new session for implicit remoting of "Get-Bios" command...
The term 'Get-Bios' is not recognized as the name of a cmdlet, function,
   script file, or operable program. Check the spelling of the name, or if
a path was included, verify that the path is correct and try again.
    + CategoryInfo          : ObjectNotFound: (Get-Bios:String) [],
   CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
    + PSComputerName        : W16TGT01
```

When you run this command, you see a message indicating that a new connection is being created. But then you get an error saying the command `Get-Bios` isn't found.

That's because you're dynamically adding the function to the remote session. When you establish a new session, because you're not adding the function, it isn't there. In the next section, we'll describe how to create remote endpoints that always contain your custom functions. There are a few other issues you need to be aware of when running commands remotely. We'll look at those next.

11.5 Considerations when running commands remotely

When you run commands on multiple computers, you need to be aware, at least to some extent, of how the execution environment can differ on the target machines. For example, the target machine may be running a different version of the operating system or it may have a different processor. There may also be differences in which applications are installed, how files are arranged, or where things are placed in the registry. In this section, we'll look at a number of these issues. Don't be put off by these issues—they're not meant to scare you. They're edge cases you need to be aware of to get the most out of PowerShell remoting.

11.5.1 Remote session startup directory

When a user connects to a remote computer, the system sets the startup directory for the remote session to a specific value. This value will change depending on the version of the operating system on the target machine. If the machine is running Windows Vista, Windows Server 2003 R2, or a later version of Windows, the default starting location for the session is the user's home directory, which is typically C:\Users\<UserName>.

On Windows Server 2003, the user's home directory is also used: C:\Documents\Settings\<UserName>. For Windows XP, the default user's home directory is used: C:\Documents\Settings\Default User.

> **NOTE** Windows Server 2003 and Windows XP are no longer supported by Microsoft and so should be less likely to be found in use with time. But from experience we can say that unsupported operating systems can easily linger for 10 years or more because of a special application that has to run on a particular version of Windows.

The default starting location can be obtained from either the `$ENV:HOMEPATH` environment or the PowerShell `$HOME` variable. By using these variables instead of hardcoded paths in your scripts, you can avoid problems related to these differences.

11.5.2 Profiles and remoting

Most PowerShell users eventually create a custom startup script or profile that they use to customize their environment. These customizations typically include defining convenience functions and aliases. Although profiles are a great feature for customizing local interactive sessions, if the convenience commands they define are used in scripts that you want to run remotely, you'll encounter problems. That's because your profiles *aren't* run automatically in remote sessions, and that means the

convenience commands defined in the profile aren't available in the remote session. In fact, the `$PROFILE` variable, which points to the profile file, isn't even populated for remote sessions.

As a best practice, for production scripting you should make sure your scripts never become contaminated with elements defined by your profiles. One way to test this is to run the script from PowerShell.exe with the `-NoProfile` option, which looks like this:

```
powershell -NoProfile -File myscript.ps1
```

This command will run the script without loading your profile. If the script depends on anything defined in the profile, it will generate errors.

But for remote interactive sessions, it'd be nice to have the same environment everywhere. You can accomplish this by using `Invoke-Command` with the `-FilePath` parameter to send your profile file to the remote machine and execute it there. The set of commands you need to accomplish this are:

```
PS> $c = Get-Credential
PS> $s = New-PSSession -Credential $c -ComputerName targetComputer
PS> Invoke-Command -Session $s -FilePath $PROFILE
PS> Enter-PSSession $s
```

First, you get the credential for the target machine (this typically won't be needed in the domain environment). Next, you create a persistent session to the remote computer. Then you use `-FilePath` on `Invoke-Command` to execute the profile file in the remote session. With the session properly configured, you can call `Enter-PSSession` to start your remote interactive session with all your normal customizations.

Alternatively, sometimes you may want to run a profile on the remote machine instead of your local profile. Because `$PROFILE` isn't populated in your remote session, you'll need to be clever to make this work. The key is that although `$PROFILE` isn't set, `$HOME` is. You can use this to compose a path to your profile on the remote computer. The revised list of commands looks like this:

```
PS> $c = Get-Credential
PS> $s = New-PSSession -Credential $ -ComputerName targetComputer

PS> Invoke-Command -Session $s {
  . "$home\Documents\WindowsPowerShell\profile.ps1" }
PS> Enter-PSSession $s
```

This command dot-sources (see section 7.1.4) the profile file in the user's directory on the remote machine into the session.

> **NOTE** This script won't work on XP or Windows Server 2003. Change the script to use `"$home\Documents and Setting\WindowsPowerShell\profile.ps1"` as the profile path.

In this section, you learned how to cause your profile to be used to configure the remote session environment. Next, we'll examine another area where these variations can cause problems.

11.5.3 Issues running executables remotely

PowerShell remoting allows you to execute the same types of commands remotely as you can locally, including external applications or executables. The ability to remotely execute commands like `shutdown` to restart a remote host or `ipconfig` to get network settings is critical for system management.

For the most part, console-based commands will work properly because they read and write only to the standard input, output, and error pipes. Commands that won't work are ones that directly call the Windows Console APIs, like console-based editors or text-based menu programs. The reason is that no console object is available in the remote session. Because these applications are rarely used any longer, this fact typically won't have a big impact. But there are some surprises. For example, the `net` command will work fine most of the time, but if you do something like this (which prompts for a password)

```
PS> net use p: '\\machine1\c$'  /user:machine1\user1 *
Type the password for \\machine1\c$:
```

in a remote session, you'll get an error:

```
[machine1]: > net use p: '\\machine1\c$'  /user:machine1\user1 *
net.exe : System error 86 has occurred.
    + CategoryInfo          : NotSpecified: (System error 86 has
       occurred.:String) [], RemoteException
    + FullyQualifiedErrorId : NativeCommandError

The specified network password is not correct.
Type the password for \\machine1\c$:
[machine1]: >
```

This command prompts for a password and returns an empty string.

The other kind of program that won't work properly is commands that try to open a user interface (also known as "try to pop GUI") on the remote computer. The program starts, but no window will appear. If the command eventually completes, control will be returned to the caller and things will be more or less fine. But if the process is blocked while waiting for the user to provide some input to the invisible GUI, the command will hang and you must stop it manually by pressing Ctrl-C. If the keypress doesn't work, you'll have to use some other mechanism to terminate the process.

One thing we can guarantee is that you'll need to access files—but when you're working remotely, how do you know which files you're using?

11.5.4 Using files and scripts

When you enter an interactive PowerShell session and access a file, such as a script or text file, you're obviously using the file on the remote machine. Remember that an interactive session is effectively like running a PowerShell session directly on the machine. But what about when you use `Invoke-Command` either directly or through a remoting session?

We're going to be running a number of commands to the remote computer (`W16TGT01`), so we'll create a remoting session:

```
PS> $s = New-PSSession -ComputerName W16TGT01
```

On the `W16TGT01` machine, a file exists with these two lines:

```
Write-Host 'Run from W16TGT01'
Write-Host $env:COMPUTERNAME
```

You know that `Invoke-Command` is used to run commands through a remoting session:

```
PS> Invoke-Command -Session $s -ScriptBlock {C:\Scripts\PiA3e\FileTest.ps1}
Run from W16TGT01
W16TGT01
```

Sometimes you may have a script on your local machine that you need to run on remote machines. One solution would be to copy the script to the remote machines and run it as in the previous example. That would be inefficient if you're dealing with hundreds or thousands of machines.

You can run a local script through a remoting session. Given a script on the local machine

```
Write-Host 'Run from W16AS01'
Write-Host $env:COMPUTERNAME
```

the `-FilePath` parameter is used to invoke a local script:

```
PS> Invoke-Command -Session $s -FilePath C:\Scripts\PiA3e\FileTest.ps1
Run from W16AS01
W16TGT01
```

Notice that the computer name that's reported is the remote machine rather than the local machine, even though you're running the script from your local disk.

One of the tenets of PowerShell remoting is isolation, but you can access local variables as well as local scripts.

11.5.5 Using local variables in remote sessions

When you use a variable in the scriptblock of a command sent to a remote machine, the assumption is that the variable is defined only in the session for the remote machine. For example, define a variable locally:

```
PS> $myvar = 123
```

Now, using the remoting session from the previous section (re-create a session if you closed that session), invoke a command using a variable with the same name:

```
PS> Invoke-Command -Session $s -ScriptBlock {"myvar is $myvar"}
myvar is
```

In the output of the command, you can see that the variable value was not made available in the remote session. In chapters 6 and 7 we discussed scope modifiers and, for instance, how you can use variables from the global scope in your functions by prefixing them with `$global:`. PowerShell remoting provides a similar (but not identical) mechanism to allow you to use local variables in remote sessions, by using the `$using:` prefix. Let's try the previous example again, but this time we'll prefix the variable with `$using:`

```
PS> Invoke-Command -Session $s -ScriptBlock {"myvar is $using:myvar"}
Myvar is 123
```

Here's what's happening: By prefixing the variable name with `$using` (introduced in PowerShell v3), you're telling PowerShell to *copy* the local value of the variable into the remote session. You're *using* the local variable in the remote session. Where this differs from scope modifiers is that it's one-way only. Changing the variable in the remote session won't change the value of the local value. In fact, if you try to change the value of the `$using` variable in the remote session, you'll get an error:

```
PS> Invoke-Command -Session $s { $using:myvar = 13 }
At line:1 char:30
+  invoke-command -localhost { $using:myvar = 13 }
+                              ~~~~~~~~~~~~
The assignment expression is not valid. The input to an assignment operator
   must be an object that is able to accept assignments, such as a variable
   or a property.
    + CategoryInfo          : ParserError: (:) [],
   ParentContainsErrorRecordException
    + FullyQualifiedErrorId : InvalidLeftHandSide
```

Now let's look at more areas where accessing the console can cause problems and how to avoid these problems.

11.5.6 Reading and writing to the console

As you saw in the previous section, executables that read and write directly to the console won't work properly. The same considerations apply to scripts that do things like call the `System.Console` APIs directly themselves. For example, call the `[Console]::WriteLine()` and `[Console]::ReadLine()` APIs in a remote session:

```
[machine1]: > [Console]::WriteLine('hi')
[machine1]: >
[machine1]: > [Console]::ReadLine()
[machine1]: >
```

Neither of these calls works properly. When you call the `[Console]::WriteLine()` API, nothing is displayed, and when you call the `[Console]::ReadLine()` API, it returns immediately instead of waiting for input.

It's still possible to write interactive scripts, but you have to use the PowerShell host cmdlets and APIs:

```
[machine1]: > Write-Host Hi
Hi
[machine1]: >
[machine1]: > Read-Host "Input"
Input: some input
some input
```

If you use these cmdlets as shown in the example, you can read and write to and from the host, and the remoting subsystem will take care of making everything work.

With console and GUI issues out of the way, let's explore how remoting affects the objects you're passing back and forth.

11.5.7 Remote output vs. local output

Much of the power in PowerShell comes from the fact that it passes around objects instead of strings. In this section, you'll learn how remoting affects these objects.

When PowerShell commands are run locally, you're working directly with the live .NET objects, which means that you can use the properties and methods on these objects to manipulate the underlying system state. The same isn't true when you're working with remote objects. Remote objects are *serialized*—converted into a form that can be passed over the remote connection—when they're transmitted between the client and the server, and *deserialized* when received by the client machine.

> **NOTE** The biggest difference you'll find is that the objects returned from a remoting session don't have any of the methods you'd have available from the same object generated locally.

Typically, you can use deserialized objects as you'd use live objects, but you must be aware of their limitations. Another thing to be aware of is that the objects that are returned through remoting will have had properties added that allow you to determine the origin of the command.

POWERSHELL SERIALIZATION

Because you can't guarantee that every computer has the same set of types, the PowerShell team chose to limit the number of types that serialize with *fidelity*, where the remote type is the same type as the local type and the object is fully re-created at the receiving end. To address the restrictions of a bounded set of types, types that aren't serialized with fidelity are serialized as a collection of properties, also called a *property bag*. This property bag has a special property, `TypeNames`, which records the name of the original type. The serialization code takes each object and adds all its properties to the property bag. Recursively, it looks at values of each the members. If the member value isn't one of the ones supported with fidelity, a new property bag is created, with members of the member's values added to it, and so on. This approach preserves structure if not the type and allows remoting to work uniformly everywhere.

DEFAULT SERIALIZATION DEPTH

The approach we have described allows any object to be encoded and transferred to another system. But there's another thing to consider: objects have members that contain objects that contain members, and so on. The full tree of objects and members can be complex. Transferring all the data makes the system unmanageably slow. This is addressed by introducing the idea of serialization depth. The recursive encoding of members stops when this serialization depth is reached. The default for objects is 1.

The final source of issues when writing portable, remotable scripts has to do with processor architectures and the operating system differences they entail. We'll work through this final set of issues in the next section of this chapter.

11.5.8 Processor architecture issues

The last potential source of problems we'll explore is the fact that the target machine may be running on a different processor architecture (64-bit versus 32-bit) than the local machine. If the remote computer is running a 64-bit version of Windows and the remote command is targeting a 32-bit session configuration, such as Microsoft.PowerShell32, the remoting infrastructure loads a Windows 32-bit process on a Windows 64-bit (WOW64) process, and Windows automatically redirects all references to the $ENV:Windir\System32 directory to the $ENV:WINDIR\SysWOW64 directory. For the most part, everything will still work (that's the point of the redirection), unless you try to invoke an executable in the System32 directory that doesn't have a corresponding equivalent in the SysWOW64 directory.

To find the processor architecture for the session, you can check the value of the `$ENV:PROCESSOR_ARCHITECTURE` variable. The following command finds the processor architecture of the session in the `$s` variable. Try this first with the 32-bit configuration:

```
PS> Invoke-Command -ConfigurationName microsoft.powershell32 `
-ComputerName localhost { $ENV:PROCESSOR_ARCHITECTURE }
x86
```

You get the expected `x86` result, indicating a 32-bit session, and on the 64-bit configuration

```
PS> Invoke-Command -ConfigurationName microsoft.powershell `
-ComputerName localhost { $ENV:PROCESSOR_ARCHITECTURE }
AMD64
```

you get `AMD64`, indicating a 64-bit configuration.

This is the last remoting consideration we're going to look at in this chapter. Don't let these issues scare you—remember, they're mostly edge cases. With some attention to detail, the typical script should have no problems working as well remotely as it does locally. The PowerShell remoting system goes to great lengths to facilitate a seamless remote execution experience. But it's always better to have a heads-up on some of the issues so you'll know where to start looking if you run into a problem.

Up to now we've been using the default remoting configuration. In the next section, we'll look at how you can create and configure your own specialized remoting configuration.

11.6 Building custom remoting services

So far, we've looked at remoting from the service consumer perspective. It's time for you to take on the role of service creator instead.

The most common remoting scenario for administrators is the one-to-many configuration, in which one client computer connects to a number of remote machines in order to execute remote commands on those machines. This is called the *fan-out* scenario because the connections fan out from a single point, and this is what you've been using in the previous sections.

In enterprises and hosted solution scenarios, you'll find the opposite configuration, where many client computers connect to a single remote computer, such as a file server or a kiosk. This many-to-one arrangement is known as the *fan-in* configuration. This mechanism is used when remote connecting to Exchange servers or Active Directory domain controllers.

Windows PowerShell remoting supports both fan-out and fan-in configurations. In the fan-out configuration, PowerShell remoting connects to the remote machine using the WinRM service running on the target machine. When the client connects to the remote computer, the WS-MAN protocol is used to establish a connection to the WinRM service. The WinRM service then launches a new process (wsmprovhost.exe) that loads a plug-in that hosts the PowerShell engine.

PowerShell remoting protocols

The transport mechanism used in PowerShell remoting consists of a five-layer stack. The stack (from top to bottom) consists of the following:

- *The PowerShell Remoting Protocol (MS-PSRP)*—https://msdn.microsoft.com/en-us/library/dd357801.aspx
- *WS-MAN (implemented by the WinRM service)*—http://mng.bz/DB74 and https://msdn.microsoft.com/en-us/library/cc251395.aspx.
- *Simple Object Access Protocol (SOAP)*—Provides an XML-based messaging framework
- HTTP and HTTPS
- TCP/IP

Creating a new process for each session is fine if there aren't many users connecting to the service. But if several connections are expected, as is the case for a high-volume service, the one-process-per-user model won't scale well. To address this issue, an alternate hosting model, targeted at developers, is available for building

custom fan-in applications on top of PowerShell remoting. Instead of using the WinRM service to host WS-MAN and the PowerShell plug-in, Internet Information Services (IIS) is used. In this model, instead of starting each user session in a separate process, all the PowerShell sessions are run in the same process along with the WS-MAN protocol engine.

Having all the sessions running in the same process has certain implications. Because PowerShell lets you get at pretty much everything in a process, multiple users running unrestricted in the same process could interfere with one another. On the other hand, because the host process persists across multiple connections, it's possible to share process-wide resources like database connections between sessions.

Given the lack of session isolation, this approach isn't intended for full-featured general-purpose PowerShell remoting. Instead, it's designed for use with constrained, special-purpose applications using PowerShell remoting. To build these applications, you need two things:

- A way to create a constrained application environment
- A way to connect to PowerShell remoting so the user gets the environment you've created instead of the default PowerShell configuration

We'll start with the second one first and look at how you specify custom remoting endpoints.

11.6.1 Working with custom configurations

When connecting to a computer by name through PowerShell remoting, the remoting infrastructure will always connect to the default PowerShell remoting service. In the non-default connection case, you also have to specify the *configuration* on the target computer to connect to. A configuration is made up of three elements:

- The name you use to connect to the endpoint
- A script that will be run to configure the sessions that will run in the endpoint
- An ACL used to control who has access to the endpoint

When using the `Invoke-Command`, `New-PSSession`, or `Enter-PSSession` cmdlets, you can use the `-ConfigurationName` parameter to specify the name of the session configuration you want to connect to. Alternatively, you can override the normal default configuration by setting the `$PSSessionConfigurationName` preference variable to the name of the endpoint you want to connect to.

When you connect to the named endpoint, a PowerShell session will be created, and then the configuration script associated with the endpoint will be executed. This configuration script should define the set of capabilities available when connecting to that endpoint. For example, there may be different endpoints for different types of management tasks—managing a mail server, managing a database server, or managing a web server. For each task, a specific endpoint would be configured to expose the appropriate commands (and constraints) required for performing that task.

11.6.2 Creating a custom configuration

Continuing our theme of remote monitoring from section 11.2.3, let's create a configuration that exposes a single custom command, `Get-PageFaultRate`. This command will return the page fault rate from the target computer.

SESSION CONFIGURATION

Every remoting connection will use one of the named configurations on the remote computer. These configurations set up the environment for the session and determine the set of commands visible to users of that session.

When remoting is initially enabled, a default configuration is created on the system called `Microsoft.PowerShell` (on 64-bit operating systems, there's also the `Microsoft.PowerShell32` endpoint). This endpoint is configured to load the default PowerShell configuration with *all* commands enabled. The security descriptor for this configuration is set so that only members of the local Administrators group can access the endpoint.

You can use the session configuration cmdlets to modify these default session configurations, to create new session configurations, and to change the security descriptors of all the session configurations. These cmdlets are shown in table 11.4.

Table 11.4 The cmdlets for managing the remoting endpoint configurations

Cmdlet	Description
`Disable-PSSessionConfiguration`	Denies access to the specified session configuration on the local computer by adding an "Everyone AccessDenied" entry to the access control list (ACL) on the configuration
`Enable-PSSessionConfiguration`	Enables existing session configurations on the local computer to be accessed remotely
`Get-PSSessionConfiguration`	Gets a list of the existing, registered session configurations on the computer
`Register-PSSessionConfiguration`	Creates and registers a new session configuration
`Set-PSSessionConfiguration`	Changes the properties of an existing session configuration
`Unregister-PSSessionConfiguration`	Deletes the specified registered session configurations from the computer
`New-PSSessionConfigurationFile`	Creates a PowerShell data language file (see module manifests) with a .pssc extension that defines a session configuration
`Test-PSSessionConfigurationFile`	Validates the contents of a session configuration file, verifying that the keys and values in the file are all valid (introduced in PowerShell v4).

REGISTERING THE ENDPOINT CONFIGURATION

Endpoints are created using the `Register-PSSessionConfiguration` cmdlet and are customized by registering a startup script. In this example, you'll use a simple startup script that defines a single function, `Get-PageFaultRate`. The script looks like this:

```
PS> @'
function Get-PageFaultRate {
 (Get-WmiObject Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
}
'@ > Initialize-HMConfiguration.ps1
```

Before you can use this function, you need to register the configuration, specifying the full path to the startup script. Call this new configuration `wpia1`. From an elevated PowerShell session, run the following command to create the endpoint:

```
PS> Register-PSSessionConfiguration -Name wpia1 `
-StartupScript $pwd/Initialize-HMConfiguration.ps1 -Force

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

Type            Keys                                Name
----            ----                                ----
Container       {Name=wpia1}                        wpia1
```

The output of the command shows that you've created an endpoint in the WSMan plug-in folder. To confirm this use (see figure 11.9), run the following:

```
PS> dir  wsman:\localhost\plugin
```

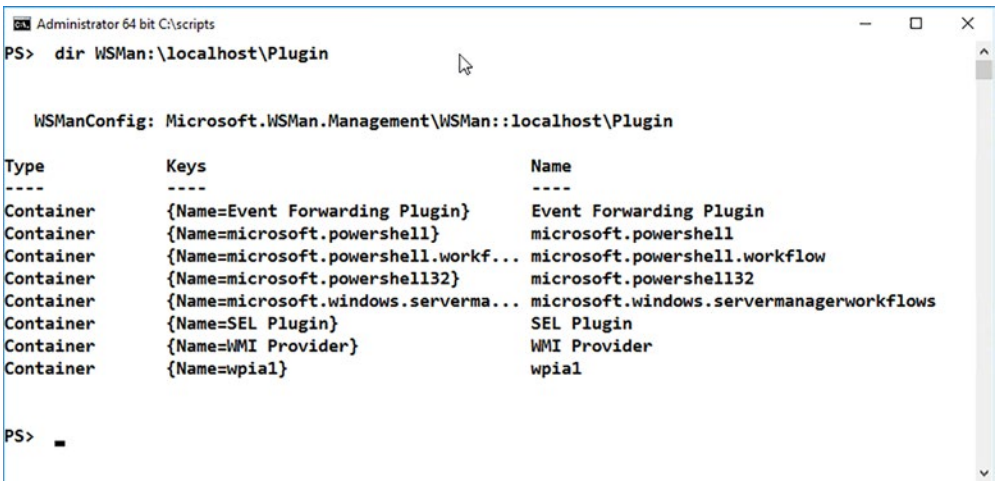


Figure 11.9 Remoting endpoints including the newly created wpia1

This shows a list of all the existing endpoints, including the one you created, `wpia1`. Now test this endpoint with `Invoke-Command` and run the function defined by the startup script:

```
PS> Invoke-Command localhost -ConfigurationName wpia1 {
Get-PageFaultRate }
68200956
```

This code verifies that the endpoint exists and is properly configured. Now clean up by unregistering the endpoint:

```
PS> Unregister-PSSessionConfiguration -Name wpia1 -Force
```

Rerun the `dir` command in figure 11.9 to verify that the endpoint has been removed.

This covers the basic tasks needed to create a custom PowerShell remoting endpoint using a configuration script to *add* additional functionality to the session defaults. Our ultimate goal, though, is to create a custom endpoint with *reduced* functionality, exposing a restricted set of commands to qualified users, so clearly, we aren't finished yet. There are two remaining pieces to look at: controlling individual command visibility, which we'll get to in a while, and controlling overall access to the endpoint, our next topic.

11.6.3 Access controls and endpoints

By default, only members of the Administrators group on a computer have permission to use the default session configurations. To allow users who aren't part of the Administrators group to connect to the local computer, you have to give those users Execute permissions on the session configurations for the desired endpoint on the target computer. For example, if you want to enable non-administrators to connect to the default remoting `Microsoft.PowerShell` endpoint, you can do so by running the following command:

```
PS> Set-PSSessionConfiguration Microsoft.PowerShell `
-ShowSecurityDescriptorUI
```

This code launches the dialog box shown in figure 11.10.

You add the name of a user or a group you want to enable to the list, then select the Execute (Invoke) check box. Then dismiss the dialog box by clicking OK. At this point, you'll get a prompt telling you that you need to restart the WinRM service for the change to take effect. Do so by running `Restart-Service winrm` as shown here:

```
PS> Restart-Service winrm
```

Once the service is restarted, the user or group you've enabled can connect to the machine using remoting.

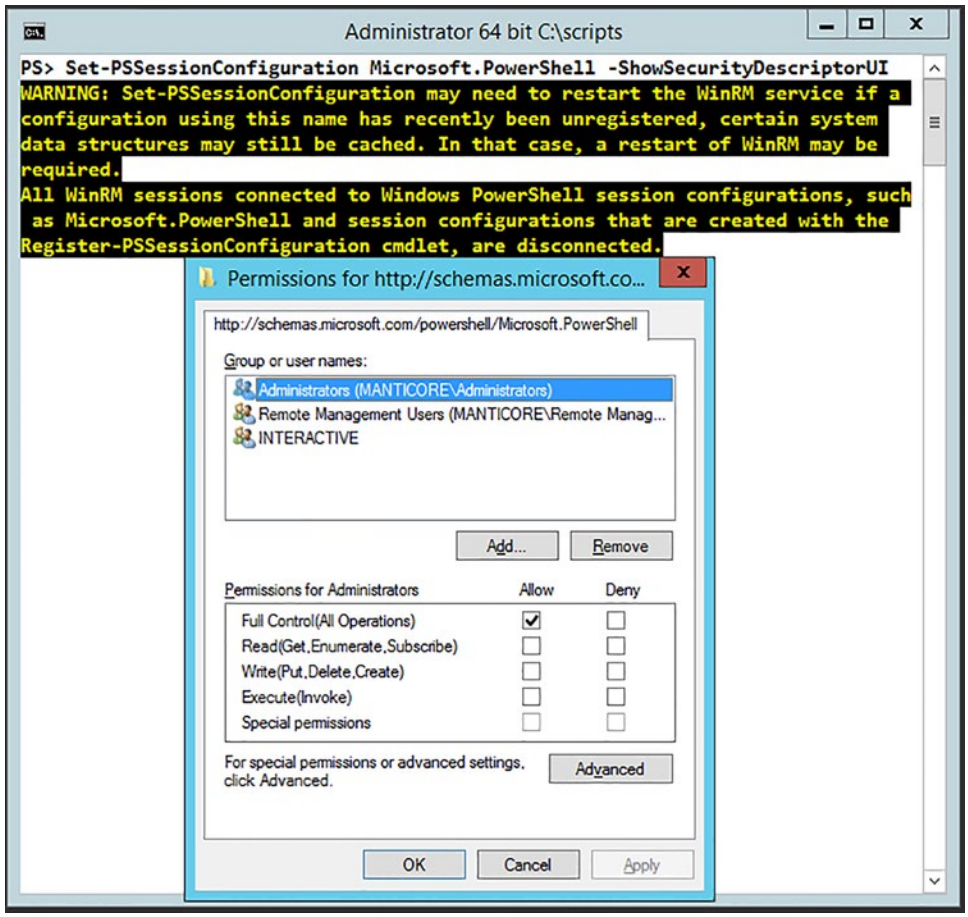


Figure 11.10 This dialog box is used to enable the Execute permission on the default remoting configuration. Use this dialog box to allow a user who isn't a member of the Administrators group to connect to this computer using PowerShell remoting.

SETTING SECURITY DESCRIPTORS ON CONFIGURATIONS

When `Enable-PSRemoting` creates the default session configuration, it doesn't create explicit security descriptors for the configurations. Instead, the configurations inherit the security descriptor of the `RootSDDL`. The `RootSDDL` is the security descriptor that controls remote access to the listener, which is secure by default. To see the `RootSDDL` security descriptor, run the `Get-Item` command as shown:

```
PS> Get-Item wsman:\localhost\Service\RootSDDL

      WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Service

Type           Name     SourceOfValue Value
----           ----     ------------- -----
System.String RootSDDL               O:NSG:BAD:P(A;;GA;;;BA)
```

```
(A;;GR;;;IU)
S:P(AU;FA;GA;;;WD)
AU;SA;GXGW;;;WD)
```

The string format shown in the `Value` output in the example uses the syntax defined by the Security Descriptor Definition Language (SDDL), which is documented in the Windows Data Types specification MS-DTYP in section 2.5.1 at http://mng.bz/QpKC.

To change the `RootSDDL`, use the `Set-Item` cmdlet in the WSMan: drive. To change the security descriptor for an existing session configuration, use the `Set-PSSession-Configuration` cmdlet with the `-SecurityDescriptorSDDL` or `-ShowSecurityDescriptorUI` parameter.

At this point, you know how to create and configure an endpoint and how to control who has access to that endpoint. But in your configuration, all you've done is add new commands to the set of commands you got by default. You haven't addressed the requirement to *constrain* the environment.

11.6.4 Constraining a PowerShell session

In section 11.6.2 you saw how to create a new remoting endpoint using `Register -PSSessionConfiguration`, and in the previous section you saw how to control who can access a particular endpoint. In this section, you'll learn how to control, or constrain, what can be done through a particular endpoint. This involves limiting the variables and commands available to the user of the session. You accomplish this by controlling command and variable *visibility*. You're creating a constrained endpoint.

The idea behind a constrained endpoint is that it allows you to provide controlled access to services on a server in a secure manner. This is the mechanism that the hosted Exchange product Outlook.com uses to constrain who gets to manage which sets of mailboxes. The mechanism can also be used in PowerShell Web Access to control access to a server and the commands that can be run on that server.

In PowerShell v2 you had to create a complex script to configure a new endpoint. The script involved manipulating the visibility of cmdlets and variables plus the definition of any new functionality you required.

In PowerShell v3 and later this task became much simpler thanks to the introduction of the `New-PSSessionConfigurationFile` cmdlet; the syntax is shown in figure 11.11.

The only required parameter is the path to the new configuration file:

```
PS> New-PSSessionConfigurationFile -Path .\Defaults.pssc
```

Configuration files are given a .pssc extension. The .pssc file structure is similar to a module manifest; it's a big PowerShell hashtable with name-value pairs. If you examine defaults.pssc (see download) produced by the example, you'll see that you can control a large number of configuration items, including these:

- Execution policy (controls which, if any, scripts can be run)
- Language mode

```
New-PSSessionConfigurationFile [-Path] <string> [-SchemaVersion <version>]
[-Guid <guid>] [-Author <string>] [-CompanyName <string>]
[-Copyright <string>] [-Description <string>]
[-PowerShellVersion <version>] [-SessionType<SessionType>]
[-ModulesToImport <Object[]>] [-Toolkits <string[]>]
[-AssembliesToLoad <string[]>] [-VisibleAliases<string[]>]
[-VisibleCmdlets <Object[]>] [-VisibleFunctions <Object[]>]
[-VisibleProviders <string[]>] [-AliasDefinitions <IDictionary[]>]
[-FunctionDefinitions <IDictionary[]>] [-VariableDefinitions <Object>]
[-RoleDefinitions <IDictionary>] [-EnvironmentVariables <IDictionary>]
[-TypesToProcess <string[]>] [-FormatsToProcess<string[]>]
[-LanguageMode <PSLanguageMode>] [-ExecutionPolicy <ExecutionPolicy>]
[-ScriptsToProcess <string[]>] [<CommonParameters>]
```

Figure 11.11 `New-PSSessionConfigurationFile` **syntax**

- Session type
- PowerShell version
- Existing aliases, cmdlets, functions, and providers that are visible in the endpoint
- New aliases, functions, and variables to create for the endpoint
- Format and type files to load and scripts to process

Language mode for a session configuration controls the types of things that can be executed in a session. The more secure you need the session to be, the more restrictive the language mode session should be. The options are shown in table 11.5.

Table 11.5 Remoting endpoint language options

Option	Meaning
`FullLanguage`	All PowerShell language elements are permitted.
`ConstrainedLanguage`	Commands that contain scripts to be evaluated are not allowed. User access is restricted to .NET framework types, objects, or methods. (This is the mode that PowerShell runs in on WinRT devices.)
`RestrictedLanguage`	Users may run cmdlets and functions. Scriptblocks aren't allowed. Only the following variables are allowed: `$PSCulture`, `$PSUICulture`, `$True`, `$False`, and `$Null`. Basic comparison operators are allowed. Assignment statements, property references, and method calls aren't permitted. (This is the language mode used in module manifests, sometimes also called *data language mode* because it can only describe data.)
`NoLanguage`	Users may run simple pipelines containing cmdlets and functions. No language elements such as scriptblocks, variables, or operators are permitted in the pipeline.

As you progress down the table, the things you can do in the endpoint become more limited until `Nolanguage`, when you're only allowed to run basic pipelines containing cmdlets and functions. The session capabilities are also controllable by restricting the

list of cmdlets and functions available to a user. For example, you can restrict the functionality of an endpoint so that a user can only reset their password in Active Directory!

The session type works in conjunction with the language mode. The session type options are listed in table 11.6.

Table 11.6 Session options for remoting endpoints

Option	Meaning	Default language mode
`Default`	Adds the Microsoft.PowerShell.Core snap-in to the session. This includes the `Import-Module` and `Add-PSSnapin` cmdlets so users can import other modules and snap-ins unless you explicitly prohibit the use of the cmdlets.	`FullLanguage`
`RestrictedRemoteServer`	Includes only the following proxy functions: `Exit-PSSession`, `Get-Command`, `Get-FormatData`, `Get-Help`, `Measure-Object`, `Out-Default`, and `Select-Object`. Use `New-PSSessionConfigurationFile` to add modules, functions, scripts, and other features to the session.	`NoLanguage`
`Empty`	No modules or snap-ins are added to the session by default. Use `New-PSSessionConfigurationFile` to add modules, functions, scripts, and other features to the session. This option is designed for you to create custom sessions by adding selected commands. If you don't add commands to an empty session, the session is limited to expressions and might not be usable.	`NoLanguage`

You can explicitly control the visibility of PowerShell elements using the `-Visible*` parameters shown in figure 11.11. This is a "white list" action. If a cmdlet or other element isn't on the list, you won't see it and therefore you won't be able to use it directly.

> **TIP** When using the `-Visible*` parameters, if you don't want to make anything visible for a particular type of command, don't use the parameter. A commented-out default value will be written to the .pssc file.

An example of an extremely constrained endpoint is provided in the following listing.

Listing 11.3 ComplexConstrainedConfiguration.ps1

```
New-PSSessionConfigurationFile `
-Path .\ComplexConstrainedConfiguration.pssc `
-Schema '1.0.0.0' `
```

```
-Author 'Richard' `
-Copyright '(c) PowerShell in Action Third Edition. All rights reserved.' `
-CompanyName 'PowerShell in Action' `
-Description 'Complex Constrained Configuration.' `
-ExecutionPolicy RemoteSigned `
-PowerShellVersion '5.0' `
-LanguageMode NoLanguage `
-SessionType RestrictedRemoteServer `
-FunctionDefinitions @{Name='Get-HealthModel';ScriptBlock={@{
            Date = Get-Date
            FreeSpace = (Get-PSDrive c).Free
            PageFaults = (Get-WmiObject `
            Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
            TopCPU = Get-Process | Sort-Object -Descending CPU
            TopWS = Get-Process | Sort-Object -Descending WS
    }};Options='None'} `
-VisibleProviders 'FileSystem','Function','Variable'
```

The execution policy is set to `RemoteSigned`, but in reality, you won't be able to run scripts, as you'll see in a while. Language mode is set to `NoLanguage` (see table 11.5) and session type to `RestrictedRemoteServer` (table 11.6). Three providers are made visible, but no modules, cmdlets, aliases, or variables are made available in the session.

A function to get the health of the system is defined and will be created when the endpoint is created. Run the script in listing 11.3 to create a configuration file. The fidelity of a configuration file can be tested:

```
PS> Test-PSSessionConfigurationFile -Path `
.\ComplexConstrainedConfiguration.pssc -Verbose
True
```

In the event of an error in the file, you will see the error only if you use the `-Verbose` parameter:

```
PS> Test-PSSessionConfigurationFile -Path .\ErrorConfiguration.pssc `
-Verbose

VERBOSE: The member 'LanguageMode' must be a valid enumeration type "System.
  Management.Automation.PSLanguageMode".
Valid enumeration values are "FullLanguage,RestrictedLanguage,NoLanguage,
ConstrainedLanguage". Change the member to the correct type in the file C:\
  MyData\PowerShellinAction3e\Code\Chapter11\ErrorConfiguration.pssc.
False
```

Creating the endpoint is performed with `Register-PSSessionConfiguration`. In the following example, any existing instances of the endpoint are removed—a useful technique when testing:

```
PS> Unregister-PSSessionConfiguration -Name wpiaccs -Force
PS> Register-PSSessionConfiguration –Path ` .\
  ComplexConstrainedConfiguration.pssc -Name wpiaccs -Force

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

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

You can see the new endpoint:

```
PS> dir WSMan:\localhost\Plugin\

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

Type       Keys                            Name
----       ----                            ----
Container  {Name=Event Forwarding Plugin}  Event Forwarding Plugin
Container  {Name=microsoft.powershell}     microsoft.powershell
Container  {Name=microsoft.powershell.w... microsoft.powershell.workflow
Container  {Name=microsoft.powershell32}   microsoft.powershell32
Container  {Name=microsoft.windows.serv... microsoft.windows.server...
Container  {Name=SEL Plugin}               SEL Plugin
Container  {Name=WMI Provider}             WMI Provider
Container  {Name=wpiaccs}                  wpiaccs
```

A remoting session can be created to the new endpoint. Notice that you have to give the name of the configuration (endpoint) that you used when performing the registration:

```
PS> $s = New-PSSession -ComputerName localhost -ConfigurationName wpiaccs
```

The session can now be used as normal. Let's start by checking the commands available:

```
PS> Invoke-Command -Session $s -ScriptBlock {Get-Command | select Name}

Name            PSComputerName RunspaceId
----            -------------- ----------
Clear-Host      localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Exit-PSSession  localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Get-Command     localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Get-FormatData  localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Get-HealthModel localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Get-Help        localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Measure-Object  localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Out-Default     localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
Select-Object   localhost      0377a4f9-5924-4cb0-83f9-a87f8a335147
```

NOTE When you look at this list of commands, you may wonder why some of them are included. For example, `Measure-Object` seems like a strange thing to have on the list. The reason these commands are included is that they're needed to implement some of the elements of the PowerShell Remoting Protocol. In particular, they're used to help with the command-discovery component described in the PowerShell Remoting Protocol Specification (MS-PSRP) section 3.1.4.5, "Getting Command Metadata."

Compare that with the results on the machine we're using to test the code for this book:

```
PS> Get-Command | Measure-Object | select Count

Count
-----
 2658
```

Our session is constrained! You'll notice that the function we defined, `Get-HealthModel`, is in the list of commands. Let's check that it works:

```
PS> Invoke-Command -Session $s -ScriptBlock {get-healthmodel}

Name                           Value
----                           -----
Date                           08/05/2017 12:57:29
TopWS                          {System.Diagnostics.Proces…
PageFaults                     146394771
FreeSpace                      67302338560
TopCPU                         {System.Diagnostics…
```

The observant reader will have noticed that we used `Get-Date` in the function, but it isn't in the list of commands we obtained from `Get-Command`. Does this mean we can use it directly even though we didn't explicitly make it visible in our configuration definition?

```
PS> Invoke-Command -Session $s -ScriptBlock {Get-Date}
The term 'Get-Date' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a
path was included, verify that the path is correct and try again.
```

And the answer is no! This is an important point to understand because it's the key to creating a restricted special-purpose endpoint: an *external* call can only access visible commands, but these commands, because they're defined as part of the configuration, can see all the other commands in the configuration. This means that an externally visible command can call any internal commands in the session. If the user makes an external call to a visible command, that visible command is able to call the private commands.

> **NOTE** All the error messages in this section will be truncated to show only the error text for brevity.

What about using it in a script block or function?

```
PS> Invoke-Command -Session $s -ScriptBlock { & {Get-Date}}
The syntax is not supported by this runspace. This can occur if the runspace
is in no-language mode.

PS> Invoke-Command -Session $s -ScriptBlock {function MyGetDate { [string]
   (Get-Date) }; MyGetDate}
The syntax is not supported by this runspace. This can occur if the runspace
is in no-language mode.
```

If you want to be able to create functions and scriptblocks, you need to be using `FullLanguage` mode in your endpoint. What about adding extra modules into the endpoint—modules provide extra functionality? Let's see what modules you have available:

```
PS> Invoke-Command -Session $s -ScriptBlock {Get-Module -ListAvailable}
The term 'Get-Module' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check thespelling of the name, or if a path
was included, verify that the path is correct and try again.
```

You can't see any modules so you can't load them because you don't know what's on the system. You might think about trying to import modules that you know are present, but it will fail. The endpoint is locked down to prevent any further functionality being imported. The function we defined as part of our configuration used variables. Can you use variables in your endpoint?

```
PS> Invoke-Command -Session $s -ScriptBlock {$x =  123; $x}
The syntax is not supported by this runspace. This can occur if the runspace
is in no-language mode.
```

No, they're not allowed. There's still a lot of functionality in legacy commands that you may think to use:

```
PS> Invoke-Command -Session $s -ScriptBlock {ping 127.0.0.1}
The term 'PING.EXE' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a
path was included, verify that the path is correct and try again.
```

Notice that the full name of the executable was recognized—but you're not allowed to run it. The final piece of functionality you may try is to run a script. You can try a simple script testch11.ps1 consisting of

```
Get-Service | Sort-Object Status
```

Try this:

```
PS> Invoke-Command -Session $s -ScriptBlock {C:\TestScripts\testch11.ps1}
The term 'C:\TestScripts\testch11.ps1' is not recognized as the name of a
cmdlet, function, script file, or operable program. Check the spelling of the
name, or if a path was included, verify that the path is correct and try again.
```

Again, the endpoint won't allow you to run anything beyond what it's been told is allowed. You do have a constrained remoting session.

> **NOTE** The example we've used is extreme but was designed to illustrate that you can create an endpoint and control exactly what functionality is exposed.

Step back and think about what you've accomplished here. With a few lines of code, you've defined a secure remote service. From the users' perspective, by using `Import-PSSession` they're able to install the contents of the session to use the services you expose—*by connecting to the service.*

Constrained sessions combined with implicit remoting results in an extremely flexible system, allowing you to create precise service boundaries with little server-side code and *no* client code. Consider how much code would be required to create an equivalent service using alternate technologies!

We'll close the chapter with a new remoting feature introduced with PowerShell v5.

11.7 PowerShell Direct

You normally use the computer name to define the remote machine for PowerShell remoting, whether you're using an interactive session, a persistent session, or `Invoke -Command` in standalone mode (no persistent session). PowerShell v5.1 supplies some new options. You can use a Hyper-V virtual machine name (not necessarily the same as the computer name) or the virtual machine ID (a GUID).

The options to use a virtual machine name or ID apply only under these circumstances:

- The virtual machine must be running on the local host.
- You must be logged on to the Hyper-V host as a Hyper-V administrator.
- You must supply valid credentials for the virtual machine—*not* domain credentials.
- The host operating system must be Windows 10, Windows Server 2016, or later.
- The virtual machine operating system must be Windows 10, Windows Server 2016, or later.

You can use the virtual machine name or ID to connect, but it's usually easier to use the name:

```
PS> Get-VM | where State -eq 'Running' |
select Name, Id

Name     Id
----     --
W16AS01  2a1eabc2-e3cd-495c-a91f-51a1ad43104c
W16DSC01 867c8460-a4fb-4785-9b7c-f27c9351db3c
W16TGT01 be4a5a3f-fc20-49f9-bb0f-b575c85e5734
```

Create a credential for the administrator account on the remote machine and then use the virtual machine name to connect:

```
PS> $cred = Get-Credential -Credential W16TGT01\Administrator
PS> Invoke-Command -VMName W16TGT01 -ScriptBlock {Get-Process} `
-Credential $cred
```

Either of these options will also work:

```
PS> Invoke-Command -VMId be4a5a3f-fc20-49f9-bb0f-b575c85e5734  `
-ScriptBlock {Get-Process} -Credential $cred

PS> Invoke-Command -VMGuid be4a5a3f-fc20-49f9-bb0f-b575c85e5734 `
-ScriptBlock {Get-Process} -Credential $cred
```

NOTE VMGuid is an alias for VMId.

You can create a persistent remoting session:

```
PS>  $s = New-PSSession -VMName W16TGT01 -Credential $cred
PS>  Invoke-Command -Session $s -ScriptBlock {Get-Process}
```

Or you can work interactively:

```
PS>  Enter-PSSession -VMName W16TGT01 -Credential $cred
[W16TGT01]: PS C:\Users\Administrator\Documents>
Use Exit-PSSession to close the interactive session.
```

There are a few things you need to remember when using PowerShell Direct:

- It's only for Hyper-V virtual machines.
- You can ignore network and firewall configurations; you're connecting over the VM bus rather than the network.
- PowerShell must be run with elevated privileges.

And with this, we've come to end of our coverage of the remoting features in PowerShell.

11.8 Summary

- Many PowerShell commands have built-in remoting using a `-ComputerName` parameter.
- Cmdlets with built-in remoting use a variety of connectivity mechanisms including DCOM and RPC.
- `Invoke-Command` uses WS-MAN for remote connectivity.
- You can create an interactive remoting session with `Enter-PSSession`.
- Interactive remoting sessions are closed with `Exit-PSSession`.
- Windows Server 2012 and later enable remoting by default. Azure IAAS virtual machines running Server 2012 R2 or higher also enable PowerShell remoting by default.
- All client operating systems and Windows Server 2008 R2 and earlier need remoting enabled by running `Enable-PSRemoting`.
- Additional configuration may be required in a non-domain environment.
- Users are authenticated using Kerberos in a domain environment when creating remoting sessions.
- Other authentication mechanisms are available for non-domain scenarios.
- `New-PSSession` is used to create a persistent remoting session.
- `Invoke-Command` and interactive sessions can use an existing session created with `New-PSSession`.
- PowerShell sessions can be disconnected and later reconnected. The reconnection can happen on the machine on which the session was created or another machine.

- You can connect to a disconnected session created by another user if you have the correct credential information.
- `Copy-Item` has `-FromSession` and `-ToSession` parameters that enable you to copy files across PowerShell remoting sessions.
- Implicit remoting enables you to import functionality from the remote system into your session. You can save the imported commands as a module.
- Profiles don't run by default in remoting sessions.
- Scripts on the local or remote machine can be run through a remoting session.
- Local variables can be accessed in a remoting session via the `$using` scope modifier.
- Custom endpoints can be created to constrain the functionality available to a user through a specific remoting connection.
- PowerShell Direct enables remoting over the VM bus from a Hyper-V host to a virtual machine on that host.

In the next chapter, we'll look at a feature introduced in PowerShell v3: PowerShell workflows.

WINDOWS ADMINISTRATION

Windows PowerShell IN ACTION

THIRD EDITION

Bruce Payette • Richard Siddaway

In 2006, Windows PowerShell reinvented the way administrators and developers interact with Windows. Today, PowerShell is required knowledge for Windows admins and devs. This powerful, dynamic language provides command-line control of the Windows OS and most Windows servers, such as Exchange and SCCM. And because it's a first-class .NET language, you can build amazing shell scripts and tools without reaching for VB or C#.

Windows PowerShell in Action, Third Edition is the definitive guide to PowerShell, now revised to cover PowerShell 6. Written by language designer Bruce Payette and MVP Richard Siddaway, this rich book offers a crystal-clear introduction to the language along with its essential everyday use cases. Beyond the basics, you'll find detailed examples on deep topics like performance, module architecture, and parallel execution.

What's Inside

- The best end-to-end coverage of PowerShell available
- Updated with coverage of PowerShell v6
- PowerShell workflows
- PowerShell classes
- Writing modules and scripts
- Desired State Configuration
- Programming APIs and pipelines

Written for intermediate-level developers and administrators.

Bruce Payette is codesigner and principal author of the PowerShell language. **Richard Siddaway** is a longtime PowerShell MVP, author, speaker, and blogger.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/windows-powershell-in-action-third-edition

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]

"This comprehensive guide to PowerShell just gets better with every revision!"
—Wayne Boaz, Nike

"Excellent coverage of the new features in PowerShell. Recommended for all levels of users."
—Lincoln Bovee, Proto Labs

"Deep technical discussions of the inner workings of PowerShell. Many useful examples. Up to date!"
—Dr. Edgar Knapp
ISIS Papyrus Europe

"If you're serious about PowerShell, you need to read this book. Seriously: Read this book!"
—Stephen Byrne, Dell

ISBN-13: 978-1-63343-029-7
ISBN-10: 1-63343-029-4
55999
9 781633 430297