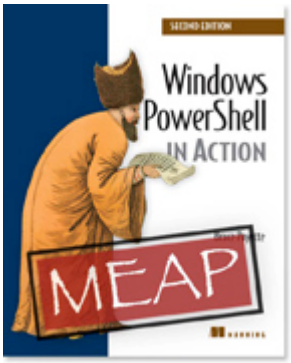
MANNING PUBLICATIONS

Implementing Multi-machine Monitoring

Excerpted from



[Windows PowerShell in Action, Second Edition]
EARLY ACCESS EDITION

Bruce Payette
MEAP Release: February 2009
Softbound print: June 2010 (est.) | 700 pages
ISBN: 9781935182139

This article is taken from the book Windows PowerShell in Action, Second Edition. *In this article, the author builds a solution for multi-machine monitoring. First, he teaches how to write a script against a single host, and then against a list of machines. Next, he describes throttling, a method of limiting the amount of resources that an activity can consume at one time. Lastly, the author shows how adding parameters can increase the flexibility of this resource management tool.*

In this article, we're going to build a solution for a real management problem – multi-machine monitoring. With this solution, we'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 when it's out of memory, out of disk, or experiencing reduced performance due to a high faulting rate. We'll gather the data on the remote host and return it as a hashtable so we can look at it locally.

Working against a single machine

To simplify things, we'll start by writing a script that can work against a single host. Listing 1 shows this script.

Listing 1 Defining the data acquisition ScriptBlock

```
$gatherInformation ={
    @{
        Date = Get-Date
        FreeSpace = (Get-PSDrive c).Free
        PageFaults = (Get-WmiObject `
            Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
        TopCPU = Get-Process | sort CPU -desc | select -first 5
        TopWS = Get-Process | sort -desc WS | select -first 5
    }
}
Invoke-Command servername $gatherInformation
```

This script uses a number of mechanisms to get the required data and returns the result as a hashtable. This hash table contains the following pieces of performance-related information:

- the amount of free space on the C drive from the `Get-PSDrive` command
- the page fault rate retrieved using WMI
- the processes consuming the most CPU from `Get-Process` with a pipeline
- the processes that have the largest working set also from `Get-Process`

We've written the information gathering scriptblock in exactly the same way we would write it for the local host – the fact that it is being remoted to another computer is entirely transparent. All we had to do was wrap the code in an extra set of braces and pass it to `Invoke-Command`.

Working against a list of machines

Working against one machine was pretty simple but our goal was *multi-machine* monitoring. So we need to change the script to run against a list of computers. We don't want to hardcode the list of computers in the script - that interferes with reuse, so we'll design it to get the data from a file called servers.txt. The content of this file is simply a list of host names, 1 per line which might look like:

```
Server-sql-01
Server-sql-02
Server-sql-03
Server-Exchange-01
Server-SharePoint-Markerting-01
Server-Sharepoint-Development-01
```

Adding this functionality only requires a small, one-line change to the call to `Invoke-Command`. This revised command looks like:

```
Invoke-Command (Get-Content servers.txt) $gatherInformation
```

We could make this more complex – say, we only wanted to scan certain computers on certain days. We'll update the servers.txt file to be a CSV file. This would look like:

```
Name, Day
Server-sql-01,Monday
Server-sql-02,Tuesday
Server-sql-03,Wednesday
Server-Exchange-01, Monday
Server-SharePoint-Markerting-01, Wednesday
Server-Sharepoint-Development-01, Friday
```

Now when we load the servers, we'll do some processing on this list which looks like:

```
$servers = Import-CSV servers.csv |
    where { $_.Day -eq (get-date).DayOfWeek } |
    foreach { $_.Name }
Invoke-Command $servers $gatherInformation
```

There are still no changes required in the actual data-gathering code. Let's move on to the next refinement.

Resource management using throttling

In a larger organization, this list of servers is likely to be quite large, perhaps hundreds or even thousands of servers. Obviously, it isn't possible to establish this many concurrent connections – it would exhaust the system resources. To manage the amount of resources consumed, we can use *throttling*. In general, throttling allows us to limit the amount of resources that an activity can consume at one time. In this case, we're limiting the number of connections that command makes at one time. In fact, there is a built-in throttle limit to prevent accidental resource exhaustion. By default, `Invoke-Command` will limit the number of concurrent connections to 32. To override this, we use the `-Throttle` parameter on `Invoke-Command` to limit the number of connections. In this example, we're going to limit the number of concurrent connections to 10 as shown:

```
$servers = Import-CSV servers.csv |
    where { $_.Day -eq (get-date).DayOfWeek } |
    foreach { $_.Name }
icm -throttle 10 $servers $gatherInformation
```

At this point, let's consolidate the incremental changes we've made and look at the updated script as a whole. This is shown in listing 2.

Listing 2 Data acquisition script using servers CSV file

```
$gatherInformation ={
```

```
        @{
            Date = Get-Date
            FreeSpace = (Get-PSDrive c).Free
            PageFaults = (Get-WmiObject `
                Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
            TopCPU = Get-Process | sort CPU -desc | select -first 5
            TopWS = Get-Process | sort -desc WS | select -first 5
        }
    }
    $servers = Import-CSV servers.csv |
        where { $_.Day -eq (get-date).DayOfWeek } |
        foreach { $_.Name }
    icm -throttle 10 $servers $gatherInformation
```

This has become a pretty capable script – it gathers a useful set of information from a network of servers in a manageable scalable way. It would be nice if we could generalize it a bit more so we could use different lists of servers or change the throttle limits.

Adding Parameterization

We can increase the flexibility of our tool by adding some parameters to it. Here are the parameters we want to add:

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

The first two are obvious – the name of the servers file and the throttle limit, both with reasonable defaults. The last one is less obvious. This parameter will control the number of process objects to include in the `TopCPU` and `TopWS` entries in the table returned from the remote host. While we could, in theory, trim the list that gets returned locally, we can't add to it so we really need to evaluate this parameter on the remote end to get full control. This means it has to be a parameter to the remote command. This is another reason why scriptblocks are useful. We can add parameters to the scriptblock that's executed on the remote end. We're finally modifying the data collection scriptblock. The modified scriptblock looks like:

```
    $gatherInformation ={
        param ($procLimit = 5)
        @{
            Date = Get-Date
            FreeSpace = (Get-PSDrive c).Free
            PageFaults = (Get-WmiObject `
                Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
            TopCPU = Get-Process | sort CPU -desc | select -first $procLimit
            TopWS = Get-Process | sort -desc WS | select -first $procLimit
        }
    }
```

and the updated call to `Invoke-Command` looks like:

```
    Invoke-Command -Throttle 10 -ComputerName $serves `
        -ArgumentList $numProcesses `
        -ScriptBlock $gatherInformation
```

Once again, let's look at the complete updated script which is shown in listing 3.

Listing 3 Parameterized data acquisition script

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

    $gatherInformation ={
```

```
    param ($procLimit = 5)
    @{
        Date = Get-Date
        FreeSpace = (Get-PSDrive c).Free
        PageFaults = (Get-WmiObject `
            Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
        TopCPU = Get-Process | sort CPU -desc | select -first $procLimit
        TopWS = Get-Process | sort -desc WS | select -first $procLimit
    }
}

$servers = Import-CSV servers.csv |
    where { $_.Day -eq (get-date).DayOfWeek } |
    foreach { $_.Name }

Invoke-Command -Throttle 10 -ComputerName $servers `
       -ArgumentList $numProcesses `
       -ScriptBlock $gatherInformation
```

This script is starting to become a bit complex. At this point, it's a good idea to separate the script code in `$gatherInformation` that *gathers* the remote information from the "infrastructure" script which *orchestrates* the information gathering. We'll put this information gathering part into its own script. We'll call this new script `BasicHealthModel.ps1` since it gathers some basic information about the state of a machine. This script is shown in listing 4.

Listing 4 Basic Health Model script

```
    param ($procLimit = 5)
    @{
        Date = Get-Date
        FreeSpace = (Get-PSDrive c).Free
        PageFaults = (Get-WmiObject `
            Win32_PerfRawData_PerfOS_Memory).PageFaultsPersec
        TopCPU = Get-Process | sort CPU -desc | select -first $procLimit
        TopWS = Get-Process | sort -desc WS | select -first $procLimit
    }
```

The orchestration code has to be changed to invoke this script. This turns out to be very easy – we can just use the -File option on `Invoke-Command` to do this:

```
Invoke-Command -Throttle 10 -ComputerName $servers `
       -ArgumentList $numProcesses `
       -Script BasicHealthModel.ps1
```

We'll put the final revised orchestration script into a file called `Get-HealthData.ps1` as shown in listing 5.

Listing 5 Data acquisition driver script

```
param (
    [parameter]
       [string] $serverFile = "servers.txt",
    [parameter]
       [int] $throttleLimit = 10,
   [parameter]
       [int] $numProcesses = 5
   [parameter]
       [int] $model = "BasicNealthModel.ps1"
)

$servers = Import-CSV servers.csv |
    where { $_.Day -eq (get-date).DayOfWeek } |
    foreach { $_.Name }

Invoke-Command -Throttle 10 -ComputerName $servers `
       -ArgumentList $numProcesses `
       -Script $model
```

This separation of concerns allowed us to add a parameter for specifying alternative health models.

The end result of all of this is that, with a very small amount of code, we've created a very flexible framework for an "agentless" distributed health monitoring system.

What we're doing here isn't really what most people would call monitoring. Monitoring 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 rather than a push model. This solution is more appropriate for configuration analysis. In fact the Microsoft Baseline Configuration Analyzer has a very similar (though much more sophisticated) architecture.

We now have an idea of how to use remoting to execute a command on a remote server. This is a powerful mechanism, but, sometimes, we need to send more than one command to a server – for example, we might want to run multiple data-gathering scripts, one after the other on the same machine. Since there is a significant overhead in setting up each remote connection, we don't want to be creating a new connection for every script we execute. Instead we want to be able to establish a persistent connection to a machine, run all of the scripts and then shut the connection down. And, that's accomplished with sessions. Check out my book for details.