Les jobs sous PowerShell.

Par Laurent Dardenne, le 30/11/2014.

Conçu avec Powershell version 3 x64 sous Windows Seven FR.

Site de l'auteur : http://laurent-dardenne.developpez.com/

Les fichiers sources :
http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/LesJobsSousPowershell/Sources.zip

Chapitres

1 Le principe

Par défaut le code Powershell exécuté dans une console est synchrone. On traite une chose à la fois et le suivi des opérations se fait de visu dans la console ou en parcourant un fichier de log.

On peut démarrer plusieurs consoles exécutant chacune un traitement différent, ici aussi la constatation des changements d'état des traitements est manuelle, on peut donc faire trois ou quatre choses en même temps avec Powershell de manière simple.

Cela nécessite tout de même une certaine dextérité au clavier et une attention particulière en tant qu'opérateur. Une fois atteint le seuil critique du suivi de traitement en simultané, qui dépend de chacun, on se posera la question de savoir comment repousser cette limite.

Une solution est de requérir l'aide d'une autre personne sur une seconde machine, celle-ci exécutera d'autres traitements. Une fois ceci fait les deux personnes vont devoir travailler ensemble afin de déterminer où elles en sont, ce qui est fait, restent à faire, ou refaire car des traitements peuvent échouer. Ce travail va demander une organisation.

On peut aussi se demander si un automatisme permettrait l'exécution et le suivi de traitement en parallèle, c'est à dire concevoir un traitement qui gère des traitements.

Depuis la version 2.0, PowerShell répond à cette question par l'introduction du concept de travail en tâche de fond (BackgroundJob). C'est-à-dire l'exécution de tâches en arrière-plan, et c'est bien ce dont on a besoin (surtout s'il y a plusieurs centaine de serveurs à traiter).

Ceci dit, que ce soit des personnes ou des automates la question de limite reste d'actualité, en effet une machine elle est aussi limitée par ses capacités.

Sous Powershell on utilise le terme de **job** au lieu de tâche de fond (un traitement en arrière-plan). On exécute un traitement de manière asynchrone et sans interaction avec la console qui elle exécute le traitement de premier plan.

Asynchrone signifie ici qu'on ne connait pas le moment où ce type de traitement se termine. On ne peut donc prévoir le point où se rejoignent le script principal (le demandeur) et le script secondaire (le réalisateur de la demande, c'est-à-dire le job) afin de récupérer le résultat du traitement, car leur **ordonnancement** ne nous appartient plus**.**

2 Cmdlets dédiés au Job

Retrouvons la liste des cmdlets natif dédiés à la manipulation des jobs :

```
$RuntimeModules=@(
 'Microsoft.PowerShell.Core', 'Microsoft.PowerShell.Diagnostics',
 'Microsoft.PowerShell.Host', 'Microsoft.PowerShell.Management',
 'Microsoft.PowerShell.Security', 'Microsoft.PowerShell.Utility',
 'Microsoft.WSMan.Management', 'ISE',
 'PSDesiredStateConfiguration', #PS v4
 'PSScheduledJob', 'PSWorkflow','PSWorkflowUtility' #PS v3
)
Get-Command -Noun *Job* -Module $RuntimeModules |
 Group-Object Noun | Select-Object Name,Count
```

```
Name                 Count
----                 -----
JobTrigger               7
ScheduledJob             6
Job                      8
ScheduledJobOption       3
```

Le groupe '***Job***' contient la liste des cmdlets de base, certains nécessitent au moins la version 3 de Powershell :

Get-Job	Obtient la liste des jobs existants déclarés dans la session.
Receive-Job	Récupère le résultat émis par le traitement associé à un job.
Remove-Job	Supprime un job.
Start-Job	Débute l'exécution d'un job sur l'ordinateur local.
Stop-Job	Arrête un job en cours d'exécution. *L'état du job passe de 'Running' à 'Stopped'.*
Wait-Job	Attend la fin d'exécution d'un job. Un temps d'attente maximum peut être précisé avant d'exécuter la commande suivante qu'un job soit terminé ou non.
Suspend-Job	Suspend l'exécution d'un job de workflow. *L'état du job passe à 'Suspended'.* **PS V3 et supérieure.** **Nécessite un job de Workflow ou un job dérivée de la classe Job2.**
Resume-Job	Reprend l'exécution d'un job de workflow suspendu. *L'état du job passe de 'Suspended' à 'Running'.* **PS V3 et supérieure.** **Nécessite un job de Workflow ou un job dérivée de la classe Job2.**

Attention, si vous utilisez PSReadline version inférieure ou égal '1.0.0.12', celle-ci contient un bug impactant la gestion des jobs d'événements.

2.1 La documentation existante

Powershell propose les fichiers d'aide suivants :

about_Jobs

about_Job_Details

about_Remote_Jobs

Compléter par ceux-ci à partir de la version 3 :

about_Scheduled_Jobs

about_Scheduled_Jobs_Advanced

about_Scheduled_Jobs_Basics

about_Scheduled_Jobs_Troubleshooting

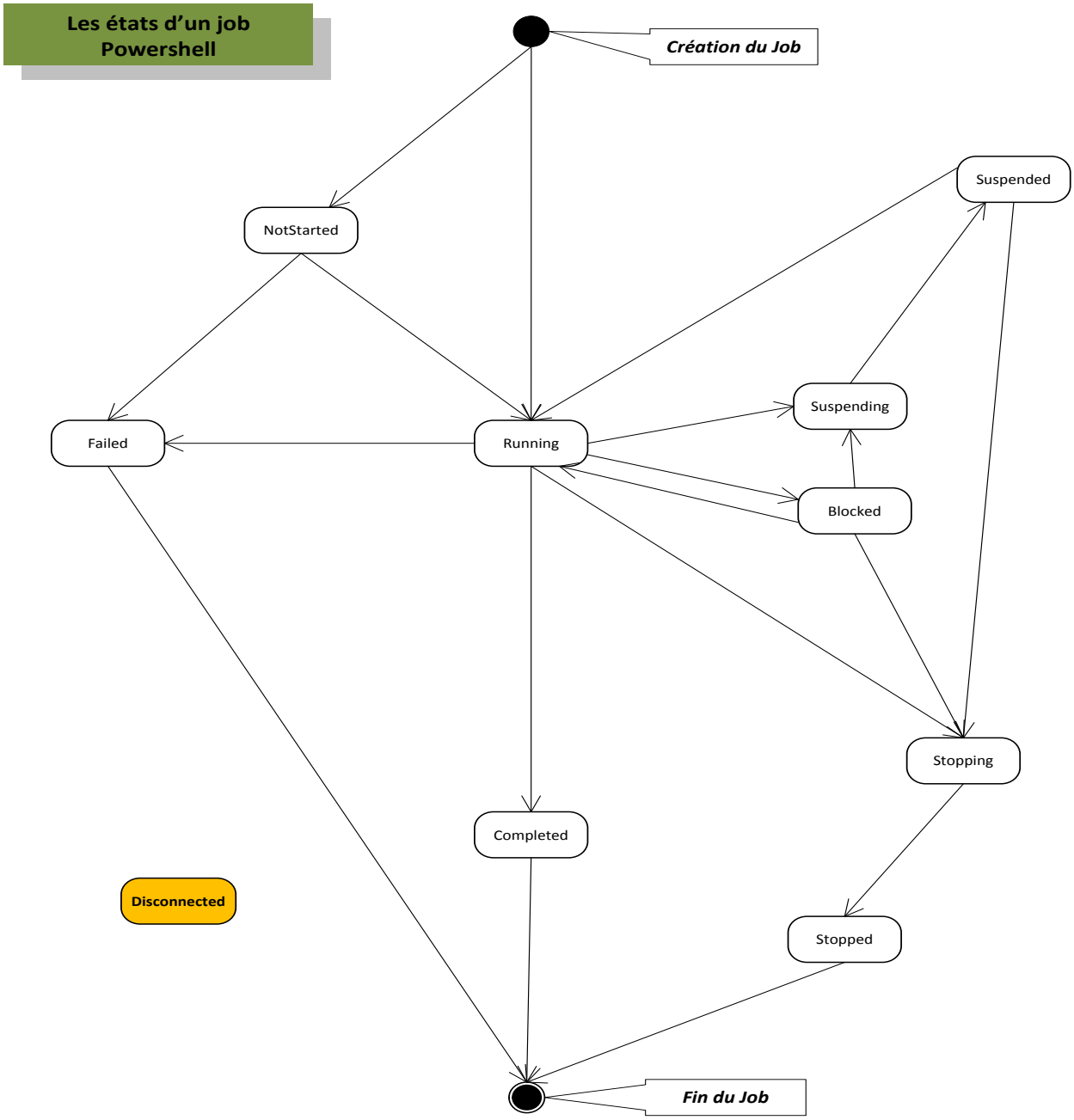
Ce tutoriel ne reprendra pas dans le détail les paramètres des cmdlets de base, reportez-vous à la documentation *via* Get-Help.

De plus, selon les versions de Powershell ceux-ci peuvent avoir des paramètres supplémentaires.

3 Les états d'un job

3.1 Etat d'un job

Avant d'aller plus loin, le diagramme suivant offre un aperçu de ma compréhension du cycle de vie d'un job Powershell.



Un objet **job** contient et exécute un traitement, une fois celui-ci terminé l'objet **job** existe toujours et peut contenir des informations renvoyées par son traitement.

3.2 Détail des états

Un job, un BackgroundJob plus précisément, exécute son traitement associé une seule fois, celui-ci peut réussir ou échouer. Pour relancer le traitement on exécute une autre instance du job avec le même code. Une fois un job créé, son état peut être un des suivants :

AtBreakpoint	**Powershell V5 :** Le job est en train d'être debuggé.
NotStarted	Le job n'a pas commencé à exécuter ses commandes.
Running	Le job est en train d'exécuter ses commandes.
Completed	Le job a terminé l'exécution de ses commandes avec succès.
Blocked	Le job est bloqué, par exemple en attente d'une saisie de l'utilisateur.
Failed	Le job n'a pas été en mesure de terminer ses commandes avec succès.
Disconnected	Le job appartient à une session distante déconnectée. La session doit être 'reconnectée' afin de connaitre son véritable état.
Stopping	Le job est en train d'arrêter le fonctionnement de ses commandes.
Stopped	Le job a été stoppé pendant l'exécution de ses commandes. Note : demander l'arrêt d'un job dans l'état '*Failed*' ou '*Completed'* n'a pas de sens.
Suspending	Le job est en train de suspendre le fonctionnement de ses commandes.
Suspended	Le job est suspendu. Il est en attente d'une reprise (*resume*).

Il n'existe pas d'état *Idle*, c'est-à-dire un ***BackgroundJob*** dont le traitement se met au repos ou en attente d'un événement.

L'état *Suspended*, interrompt momentanément le job, le traitement associé ne peut pas déclencher la sortie de cet état puisqu'il est en suspens, seul un traitement externe peut le faire.

De plus l'état *Suspended* ne concerne pas les ***BackgroundJob***, mais une catégorie particulière de job dont ceux de type *Workflow*. Demander la suspension d'un autre type de job déclenchera une erreur.

Un appel au cmdlet **Start-Sleep** dans le code du traitement ne modifie pas l'état du job associé, seul son traitement est momentanément suspendu.

Un job lié à un abonnement d'événement est créé dans l'état *NotStarted*, la réception du premier événement modifie son état en *Running.* Cet état demeure, tant que l'abonnement est actif.

Un job est considéré comme terminé (***Finished***) s'il est dans l'un de ces états :

Completed, Failed, Stopped.

Il est inopérant d'attendre un job dans l'état '*Stopped*', d'en stopper un dans l'état '*Completed*', ni d'en suspendre un dans l'état '*Failed'*.

L'état *Disconnected* est particulier car il peut être précédé et suivi par de nombreux états. Pour faciliter la lecture du diagramme précédent je ne les ai pas indiqués.

L'état *Disconnected* est lié à l'état d'une session distante, si on la déconnecte, il est dès lors impossible de connaitre les changements d'état des jobs de cette session qui continuent leur déroulement.

En revanche lorsque l'on s'y reconnecte, l'état des jobs distants est mis à jour. Il est donc possible que le changement d'état soit l'état *Running* si le traitement du job n'est pas terminé.

Le script *"\Sources\Session disconnected.ps1"* permet de visualiser ces changements d'état.

Pour en revenir à notre job de base et dans le cas où tout se passe correctement, le cycle d'un BackgroundJob est le suivant :

Running -> Completed

4 Utiliser un job local

Il s'agit ici des jobs exécutés sur une seule et même machine.

4.1 Exécution d'un BackgroundJob

Pour démarrer un job local on utilise le cmdlet **Start-Job** en lui associant un traitement :

```
$Job=Start-job –ScriptBlock {Write-Host "Affiche un message"}
```

Note : on peut aussi utiliser le paramètre -FilePath en lieu et place du paramètre –ScriptBlock, mais sachez que dans ce cas la variable $MyInvocation n'est pas complétement renseignée.

Dès lors le cycle commence, le cmdlet renvoie un objet job qui possède un identifiant unique :

```
$Job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
--	----	-------------	-----	-----------	--------	-------
2	Job2	BackgroundJob	Running	True	localhost	Write-Host "Affiche un...

La propriété *ID* permet de retrouver ce job dans le référentiel de jobs. Une fois l'instance du job créée, la propriété *State* nous indique son état. Ce job est en cours d'exécution.

On consulte l'objet soit par la variable $Job soit par l'appel au cmdlet **Get-Job** en lui précisant l'ID du job :

```
$Job
#ou Get-Job -id 2
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
--	----	-------------	-----	-----------	--------	-------
2	Job2	BackgroundJob	Completed	True	localhost	Write-Host "Affiche un...

Son état est *Completed*, le traitement est donc terminé, on peut désormais récupérer son résultat.

Attention, si le paramètre ***–Name*** est absent, un nom généré lui est attribué. Le nom n'est pas unique alors que le numéro de l'ID l'est.

Note :

La propriété *Command* permet au BackgroundJob de connaitre le code de son traitement, mais une fois celui-ci démarré il ne connait rien de son créateur, il n'existe pas de variable automatique portant des informations sur son parent, bien qu'il soit possible de rechercher l'*Id* du processus parent.

On peut toutefois différencier l'environnement d'exécution en cours en interrogeant la variable ***$Host.Name*** qui contient dans ce cas '*ServerRemoteHost*' au lieu de '*ConsoleHost*'.

Un BackgroundJob ne semble pas débuter son cycle par l'état *NotStarted*, ou de manière furtive, il passe directement à l'état *Running*.

4.2 Récupérer le résultat d'un Job

Dans l'exemple précédent le traitement devrait afficher un message sur la console, on constate qu'il n'en est rien.

L'objectif d'un job étant d'exécuter un traitement en arrière-plan, il ne peut interagir avec l'avant-plan, notre console.

La propriété *HasMoreData* d'un objet job indique la présence d'informations à récupérer.
L'objet job ne propose pas de propriété ou méthode pour récupérer le résultat de son traitement, on doit utiliser le cmdlet **Receive-Job** :

```
Receive-Job $job
```

```
Affiche un message
```

Le traitement fait bien ce qu'on lui demande de faire. En revanche, c'est le cmdlet **Receive-Job** qui déclenche l'affichage. Si on essaie une seconde fois de récupérer l'information :

```
Receive-Job $job
```

Rien ne se passe, car il n'y plus de données à récupérer :

```
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
2      Job2            BackgroundJob   Completed     False           localhost            Write-Host "Affiche un...
```

L'usage du paramètre *–Keep* consomme les données du job, mais ne les supprime pas :

```
$Job=Start-job -ScriptBlock {Write-Host "Affiche un message"}
Receive-Job $job -Keep
```

```
Affiche un message
```

```
Receive-Job $job -Keep
```

```
Affiche un message
```

Il reste possible d'appeler ce cmdlet sur un job en cours d'exécution.

Le résultat possède une propriété *PSComputerName* contenant le nom de la machine locale. Celle-ci est identique au contenu de la propriété *Location* du job parent.

Note : un job démarre toujours avec sa propriété *HasMoreData* à $True et celle-ci peut être à $False une fois le job terminé et ce avant d'avoir appelé **Receive-Job**.

4.2.1 Les flux à réceptionner

Un job propose les propriétés suivantes associées aux flux Powershell correspondant :

Debug, Error, Output, Progress, Verbose, Warning.

On peut lire ces flux directement :

```
$Job=Start-Job -Name "VisuEtat" -ScriptBlock {
   $VerbosePreference='continue'
   1..3|Foreach {Write-verbose "Test_$_"}
}
$Job.ChildJobs[0].Verbose
```

```
Test_1
Test_2
Test_3
```

L'appel à **Receive-Job** reçoit tous ces flux sous réserve que les variables de préférence associées soient configurées dans le code du job.

4.2.2 APIs d'affichage

Receive-Job récupère les données issues des flux Powershell uniquement :

```
$action={
 [System.Console]::WriteLine('Affichage via les APIs du framework dotNet')
 tzutil /g
 Write-host 'Affichage via les APIs de Powershell'
 }
start-job -Name 'Job1' -ArgumentList 'Server1','Job1' -ScriptBlock $action
Get-Job|Receive-job
```

```
Romance Standard Time
Affichage via les APIs de Powershell
```

L'affichage d'un programme console fonctionne, mais l'appel aux méthodes d'affichage *via* la classe *System.Console* n'est pas pris en compte dans le script.

Notez que les appels à **Clear-Host** seront exécutés en local.

4.3 Attendre la fin d'un job

Le plus souvent on souhaite attendre la fin du job avant de récupérer ses données, le cmdlet **Wait-Job** répond à ce besoin :

```
Wait-Job $job
```

Ce cmdlet est synchrone, et bien que les jobs continuent leur exécution, celui-ci bloque l'exécution du script courant. Dans la console la saisie de *Control-C* stoppe l'attente de **Wait-Job**. A moins d'utiliser un time out sur un traitement ou pour l'isoler, on peut se demander pourquoi utiliser un seul job si c'est pour se retrouver à attendre sa fin d'exécution.

Si on exécute plusieurs jobs, cette question ne se pose plus. Utiliser sans paramètre, **Wait-job** attend que TOUS les jobs dans l'état '*Running*' changent d'état, puis renvoie chaque job concerné par ce changement d'état.

Attendre un job dans l'état '*Blocked*' déclenchera une exception :

```
$Job=Start-job -Name 'LDJob' -ScriptBlock {read-Host "Message"}
Wait-Job $job
```

```
Wait-Job : Impossible de terminer l'applet de commande Wait-Job, car une ou plusieurs tâches sont bloquées
dans l'attente de l'intervention de l'utilisateur. Traitez la sortie de tâche interactive à l'aide de l'applet de
commande Receive-Job et réessayez.
```

A moins de préciser le paramètre -*State*, l'existence dans la liste des job traités d'un job à l'état *'NotStarted'* bloquera **Wait-Job** jusqu'à ce que ce job bascule dans l'état *'Running'*. Dès lors l'attente effective débutera.

Les jobs d'événement restant dans l'état '*Running*' sont pris en compte et bloqueront à leur tour l'appel à **Wait-Job**. Pour régler ce problème une norme de nommage des jobs est nécessaire :

```
$Job=Start-job –Name 'LDJob' –ScriptBlock {Write-Host "Message"}
Wait-Job –Name LDjob*
```

Ainsi on ne traite qu'une partie des jobs en cours, sans avoir à mémoriser les id des jobs, ni leurs types.

On peut utiliser le paramètre ***–Any***, dans ce cas le premier job parent changeant d'état stoppe l'attente de **Wait-Job**. On doit réitérer son exécution pour traiter les jobs restant à surveiller.

Le paramètre ***–State*** permet d'attendre un changement d'état pour ceux étant dans un état particulier. Dans l'exemple suivant on attend qu'un des jobs en cours d'exécution change d'état :

```
wait-job -state 'Running'
```

On retrouve notre problème de jobs d'événement, mais dans ce cas on ne peut pas utiliser le paramètre ***-Name*** :

```
wait-job -state 'Running' –Name LDjob*
```

```
Wait-Job : Le jeu de paramètres ne peut pas être résolu à l'aide des paramètres nommés spécifiés.
```

Ni le pipeline :

```
Get-job -Name Visu*|wait-job -state 'Running'
```

```
wait-job : L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette commande
n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun des
paramètres qui acceptent l'entrée de pipeline.
```

Je reste dubitatif sur ces possibilités autorisées indiquées dans l'aide :

```
wait-job -state 'Completed'
wait-job -state 'Failed'
wait-job -state 'Stopped'
```

Car si j'ai bien compris, on demande d'attendre un changement d'état sur un job terminé qui ne changera plus d'état.

Ici **Wait-Job** retourne immédiatement le ou les jobs dont l'état correspond à celui demandé ou bien $Null s'il n'en existe aucun. Ce type d'appel me semble similaire à celui-ci :

```
Get-Job -State Stopped # ou 'Completed' ou 'Failed'
```

Ceci dit, le comportement reste cohérent ☺

4.3.1 Waiting for the sun

Les jobs d'événement ne sont pas sensibles au soleil, mais aux boucles d'attente.

Sachez que le cmdlet **Wait-Job** suspend le traitement des événements, les événements restent dans la file d'attente et sont traités une fois la boucle d'attente terminée et ceci afin qu'il n'y ait qu'un seul traitement qui puisse modifier l'état de session.

Soyez attentif également aux cmdlets proposant un paramètre *–Wait*, par exemple **Get-Content**, et aux méthodes du même type telle que celle-ci :

```
$service = Get-Service -Name Spooler
$service.WaitForStatus('Stopped','00:01:0')
```

4.4 Supprimer un job

Une fois le résultat du traitement d'un job consommé, il est préférable de rapidement supprimer l'objet job en utilisant le cmdlet **Remove-Job** :

```
Remove-Job $job
```

Un contrôle sur la propriété *HasMoreData* est préférable :

```
Get-Job | Where-object {-not $_.HasMoreData} | Remove-Job
```

Attention à ne pas supprimer les jobs d'événements si vous utilisez le paramètre *–Force* :

```
Get-Job –Name LDjob*|
 Where-object {-not $_.HasMoreData} |
 Remove-Job -Force
```

Ce cmdlet appel en interne la méthode *Dispose()* sur chaque objet job.

4.5 Steve, je suis ton père.

Dans la seconde édition de Powershell in action, Bruce Payette indique que les jobs locaux utilisent un mécanisme de communication différent des jobs distants, ce qui fait que la configuration du remoting n'est pas un prérequis à l'utilisation des jobs locaux.

La session parente et le job enfant local utilisent des pipes anonymes pour communiquer (IPC). Un BackgroundJob est donc toujours constitué d'un parent et d'un seul enfant.

Pour les jobs distants la communication se fait *via* WinRM, un RemoteJob peut être constitué d'un parent et d'un ou plusieurs enfants.

Le job parent supervise l'exécution des jobs enfant et chacun de ses jobs peut être adressés individuellement.

La collection des jobs enfants est stockée dans la propriété *ChildJobs* du job parent, celui renvoyé par le cmdlet **Start-Job ou Invoke-Command** :

```
$Job=Start-job -ScriptBlock {Write-Host "Affiche un message"}
Wait-Job $Job
$Job.ChildJobs
```

```
Id   Name    PSJobTypeName   State       HasMoreData   Location    Command
--   ----    -------------   -----       -----------   --------    -------
2    Job2    BackgroundJob   Completed   True          localhost   Write-Host "Affiche un...
```

```
$Job.ChildJobs
```

```
Id   Name    PSJobTypeName   State       HasMoreData   Location    Command
--   ----    -------------   -----       -----------   --------    -------
3    Job3                    Completed   True          localhost   Write-Host "Affiche un...
```

```
Receive-Job $job
```

```
Affiche un message
```

```
Get-Job $Job.Id –IncludeChildJob
```

```
Id   Name    PSJobTypeName   State       HasMoreData   Location    Command
--   ----    -------------   -----       -----------   --------    -------
2    Job2    BackgroundJob   Completed   False         localhost   Write-Host "Affiche un...
3    Job3                    Completed   False         localhost   Write-Host "Affiche un...
```

Le cmdlet **Invoke-Command** permet en un seul appel l'exécution d'un même traitement sur plusieurs machines en lui passant en paramètre une liste de noms de serveur, nous aurons donc un parent et plusieurs enfants (un job par serveur).

«La tâche parent représente toutes les tâches enfants. Lorsque vous gérez une tâche parent, vous gérez également les tâches enfants associées. Par exemple, si vous arrêtez une tâche parente, toutes les tâches enfants sont arrêtées. Si vous obtenez les résultats d'une tâche parent, vous obtenez les résultats de toutes les tâches enfants».

5 Un peu de plomberie

Pour comprendre quelques points abordés plus avant, étudions sommairement ce qui se passe lors du démarrage d'un job local. On utilisera WMI pour surveiller la création et la suppression de processus. Vous pouvez utiliser le script *'\Event\SurveillanceProcessWMI.ps1'*.

5.1 Création

Pour cette surveillance on s'abonne à un événement du référentiel WMI, cet abonnement crée automatiquement un job d'un type particulier, il reste en attente :

```
$Query=@"
  Select * from __InstanceCreationEvent within 5
   where targetinstance isa 'Win32_Process'
"@
$Action = {
   $Target=$event.SourceEventArgs.Newevent.TargetInstance
   $S="Process créé {0}\{1} : {2}"
   $msg= $S -F $Target.ExecutablePath,$Target.Name,$Target.CommandLine
   Write-Warning $msg
}#$Action

Register-WMIEvent -query $Query -sourceIdentifier "TraceProcess" -action
$Action > $Null
```

Une fois ceci fait, on lance un BackgroundJob qui affiche quelques informations :

```
Start-Job -Name VisuProcess -ScriptBlock {
  Write-Warning "Type de process = $Env:PROCESSOR_ARCHITECTURE"
  $MT=[System.Threading.Thread]::CurrentThread.GetApartmentState()
  Write-Warning "Modéle de thread=$MT"
  Write-Warning "PS version=$($PSVersionTable.PSVersion)"
  Write-Warning "Process id =$PID"
  Write-Warning "Runspace id =$([Runspace]::DefaultRunspace.InstanceId)"
  Sleep -s  5
}| Wait-Job| Receive-Job
 $PID
```

Le cmdlet **Receive-Job** déclenche l'affichage des appels au cmdlet **Write-Warning** :

```
AVERTISSEMENT : Type de process = AMD64
AVERTISSEMENT : Modèle de thread=MTA
AVERTISSEMENT : PS version=3.0
AVERTISSEMENT : Process id =5328
AVERTISSEMENT : Runspace id =62f39ce1-c41a-42d5-b78e-7d53a65a804a
AVERTISSEMENT : Process créé C:\Windows\system32\conhost.exe\conhost.exe : …
AVERTISSEMENT : Process créé
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe\powershell.exe :…
```

On constate qu'un second processus Powershell est exécuté, en tant que processus enfant, et que l'identifiant du processus de la console Powershell est bien différent de celui affiché dans le job :

```
$pid
```

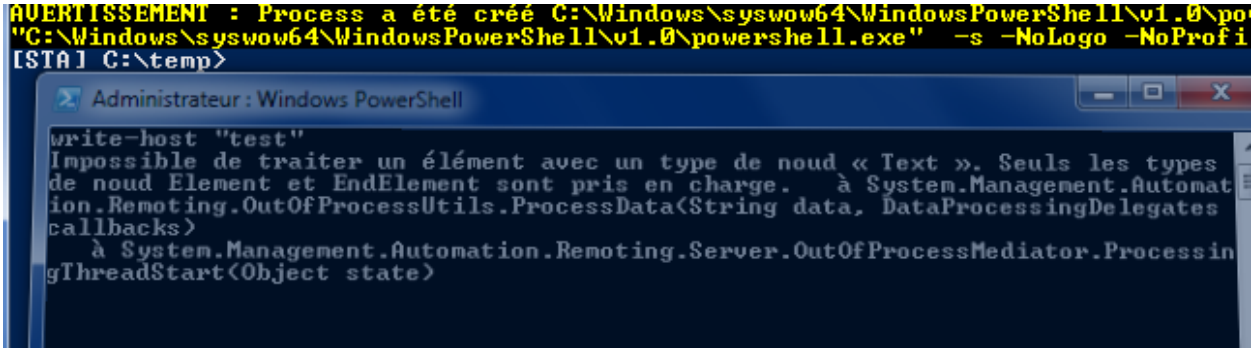
```
5468
```

Note : le processus conhost.exe est associé aux programmes console.

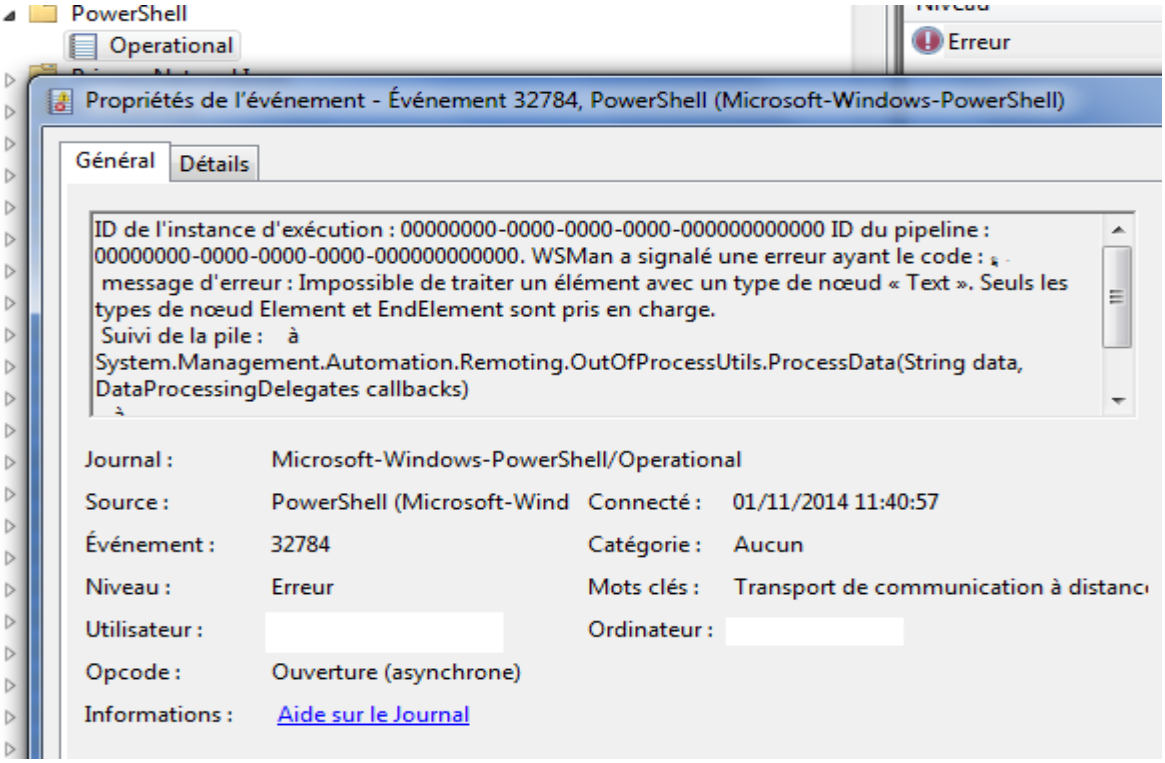
Voici la ligne d'appel du second processus :

```
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -Version 3.0 -s -NoLogo -NoProfile
```

Le paramètre ***–s*** qui n'est pas documenté, crée une console dans un mode particulier, où la saisie d'une instruction provoque une exception :



Dans les traces Powershell on trouve ce message :



Si on recherche avec ILSpy la classe citée dans le message de l'exception, on peut y lire ceci :

```
internal static void ProcessData(string data, OutOfProcessUtils.DataProcessing
{
    if (string.IsNullOrEmpty(data))
    {
        return;
    }
    XmlReader xmlReader = XmlReader.Create(new StringReader(data), OutOfProces
    while (xmlReader.Read())
    {
        XmlNodeType nodeType = xmlReader.NodeType;
        if (nodeType != XmlNodeType.Element)
        {
            if (nodeType != XmlNodeType.EndElement)
            {
                throw new PSRemotingTransportException(PSRemotingErrorId.IPCUn
                {
                    xmlReader.NodeType.ToString(),
                    XmlNodeType.Element.ToString(),
                    XmlNodeType.EndElement.ToString()
                });
            }
```

Je suppose que dans cet état la console attend des commandes SOAP (format en XML) …

5.2 Suppression

Pour vérifier la suppression dupliquer le code précédent en modifiant le nom de l'événement WMI '***__InstanceCreationEvent'*** par '***__InstanceDeletionEvent***' :

```
AVERTISSEMENT : Process détruit C:\Windows\system32\conhost.exe\conhost.exe
AVERTISSEMENT : Process détruit
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe\powershell.exe
```

Une fois le job terminé et son état mis à jour, le processus Powershell l'ayant créé, ici la console, supprime les processus enfants associés. Les données du job restent accessibles en local.

Si on supprime la session Powershell, tous les processus enfants sont détruits automatiquement.

6 Autres points à considérer

Ce que l'on a vu jusqu'à maintenant est assez simple et suffisant pour utiliser des jobs.

En revanche pour le suivi, c'est à dire le traitement de surveillance des traitements en tâche de fond, il est nécessaire de connaitre certains aspects de Powershell.

La difficulté d'apprentissage n'est pas liée aux jobs, mais aux connaissances nécessaires à la construction d'un traitement élaboré. Si toutefois vous souhaitez associer ces différentes techniques.

6.1 Passage de paramètres

Voici quelques possibilités pour paramétrer un job :

```
$env:MyPath="C:\Temp\Demo1"
$JobName='JobTest'
Start-Job -ArgumentList $JobName -Name $Jobname -ScriptBlock {
  param($Jobname)
    Write-Warning "$JobName"
    Write-Warning "$env:MyPath"
 }|Wait-Job|Receive-Job
```

```
AVERTISSEMENT : JobTest
AVERTISSEMENT : C:\Temp\Demo1
```

Dans cet exemple le paramètre *-ArgumentList* contient le nom du job associé au traitement, on peut le préciser pour nommer un fichier de logs. Si la clause ***Param()*** n'est pas précisée dans le code du scriptblock, adressez la variable automatique ***args***.

Comme nous l'avons vu précédemment, le cmdlet **Start-Job** crée un processus enfant, il est donc possible de créer une variable dans le provider d'environnement du parent et la récupérer dans le provider d'environnement de l'enfant, puisque les variables d'environnement sont propagées lors de la création d'un processus enfant.

La base de registre peut également servir à héberger un paramétrage :

```
$RegistryKey="registry::\HKEY_USERS\.DEFAULT\Software\FIRME\Traitement1"
$myPath=(Get-ItemProperty $RegistryKey).InstallPath
Set-Location "$MyPath\Scripts"
. .\Initialize-Server.ps1 –parameter1 –parameter2
Start-Task $Mode $NomMachine
…
```

6.1.1 Portée Using :

La version 3 de Powershell ajoute un scope nommé *using*, celui-ci facilite l'usage de variable locale dans le code de traitement du job :

```
 #Référence l'objet courant d'un pipeline
"Data1"," Data2"| foreach { Start-Job -ScriptBlock { "Name $using:_ " } }|
 Wait-Job|Receive-Job
```

```
Name  Data1
Name  Data2
```

```
$Path='C:\Temp\Datas'
$File=get-item $PShome
Start-Job -ScriptBlock {
 "path using =$using:path"

 $path= 'X:\'
 "path local redéclaré =$path"
 "path using =$using:path"

 $using:File|gm
}|wait-job|Receive-Job
```

```
path using =C:\Temp\Datas
path local redéclaré =X:\
path using =C:\Temp\Datas

   TypeName : Deserialized.System.IO.DirectoryInfo …
```

Le résultat nous indique que la variable *$using:Path* référence bien celle de l'appelant, le nom de variable *Path* peut être réutilisé et l'objet fichier est désérialisé, ce n'est pas l'objet fichier en lui-même, mais une représentation (*Deserialized*).

En fait Powershell transforme les occurrences de *$Using:* en une création de variable dans le contexte du traitement en la préfixant de '__using_' :

```
// System.Management.Automation.Language.UsingExpressionAst
internal const string UsingPrefix = "__using_";
```

Vérifions :

```
$Path='C:\Temp\Datas'
Start-Job –Argumentlist 'Srv1' -ScriptBlock {
  Param($Server)
   "path using =$using:path"
    Dir variable:__using_*
      Write-host "paramètre Server=$Server"
    $PSBoundParameters
   $args.count
}|wait-job|Receive-Job
```

```
path using =C:\Temp\Datas

Name                           Value
----                           -----
__using_path                   C:\Temp\Datas
paramètre Server=Srv1

Key    : __using_path
Value  : C:\Temp\Datas
Name   : __using_path

Key    : Server
Value  : Srv1
Name   : Server

0
```

Ici la variable ***$PSBoundParameters*** contient la liste des variables liées au scriptblock du traitement, celles de la clause *Param* et celle de l'occurrence du mot clé *$Using:*.

La collection *$args* est vide. Sans la clause *Param* la collection *$args* est renseignée :

```
$Path='C:\Temp\Datas'
Start-Job –Argumentlist 'Srv1' -ScriptBlock {
   "path using =$using:path"
    Dir variable:__using_*
    Write-host "paramètre Server=$($args[0])"
   $PSBoundParameters
   $args.count
}|wait-job|Receive-Job
```

```
path using =C:\Temp\Datas

Name                           Value
----                           -----
__using_path                   C:\Temp\Datas
paramètre Server=Srv1

Key    : __using_path
Value  : C:\Temp\Datas
Name   : __using_path

1
```

6.1.2 PSSenderInfo

Cette variable automatique contient des informations concernant le compte ayant exécuté la session Powershell :

```
$PSSender=Start-Job -ScriptBlock {$PSSenderInfo}|wait-job|Receive-Job
$PSSender|fl *
```

```
ConnectedUser        : Server\Account
RunAsUser            : Server\Account
PSComputerName       : localhost
RunspaceId           : df32c82f-7de4-46f8-8dc2-6993f8625000
```

```
PSShowComputerName   : False
UserInfo             : System.Management.Automation.Remoting.PSPrincipal
ClientTimeZone       : System.CurrentSystemTimeZone
ConnectionString     : http://localhost
ApplicationArguments : {PSVersionTable}
```

Consultez le détail dans l'aide nommée *about_automatic_variable*.

6.2 Initialisation du traitement d'un job

Pour des traitements nécessitant de nombreux prérequis (variables, scripts, fonctions,…) on peut utiliser un script d'initialisation. Dans ce cas l'organisation de ces différents éléments est importante et préalable au développement de votre traitement en tâche de fond.

L'usage d'un module simplifie cette organisation, car il propose la sienne. La variable d'environnement *%PSModulePath%* peut être configurée dans la zone système ou en local dans le traitement principal, le nouveau contenu temporaire de cette variable d'environnement sera propagé aux jobs. N'oubliez pas d'envisager le versioning de module…

Dans le cas de job distants, ces scripts devront être copiés sur chaque machine ou accessibles sur un partage.

6.2.1 InitializationScript

On peut aussi préparer la session avant l'exécution du job, par exemple configurer des variables de préférence ou charger des fonctions, à l'aide du paramètre *-InitializationScript* :

```
[Runspace]::DefaultRunspace.InstanceId

$SbInitJob={Write-Warning "[Initialisation] Runspace id =$(
[Runspace]::DefaultRunspace.InstanceId)"}

$sbTraitement={"Workaround">$null;Write-host "[Traitement] Runspace id =$(
[Runspace]::DefaultRunspace.InstanceId)" }

Start-job -InitializationScript $SbInitJob -scriptblock $sbTraitement|
 wait-job| Receive-Job
```

Vous constaterez à l'affichage que les GUID des runspaces distants sont identiques pour le code des paramètres *-InitializationScript* et *-Scriptblock.*

Pour injecter des fonctions locales dans le code du job on peut procéder ainsi :

```
function FonctionLocale { write-host "FonctionLocale" }

$FnctList=@("FonctionLocale")
 push-Location; set-Location Function:
   $ScriptText=$FnctList|
     Get-Item | #Where {$_.ModuleName -eq [string]::Empty}
     Foreach {
       "{0} {1} {{`r`n{2}}}`r`n" -F $_.CommandType,$_.Name,$_.Definition
     }
 pop-Location

$SbInitJob=$ExecutionContext.InvokeCommand.NewScriptBlock($ScriptText)

Start-Job -InitializationScript $SbInitJob -ScriptBlock {FonctionLocale}|
 Wait-Job| Receive-Job
```

Si une des fonctions est hébergée dans un module, il est préférable de l'utiliser *via* son module.

6.3 Profondeur de sérialisation

La sérialisation permet de convertir un objet en mémoire dans une autre structure le décrivant, et ce afin de le manipuler dans un autre contexte. L'inverse étant la désérialisation qui, à partir des informations reçus, créera soit un clone de l'objet, un autre objet similaire, soit une représentation de l'objet, souvenez-vous que « *la carte n'est pas le territoire* ».

Comme nous l'avons vue rapidement dans le chapitre sur la portée *Using:*, lors du transfert PowerShell sérialise ses données en utilisant le XML.

L'exemple suivant construit un objet personnalisé imbriquant des objets :

```
$Result = Start-job {
                $Objet1=New-Object PSobject -property @{
                        Level1=1..5
                        Level2=@((6..10),(11..15))
                        Level3=@(@((16..20),(21..25)),@((26..30)))
                      }
                $Objet2=New-Object PSobject -property @{
                        Nested=$Objet1
                        Process=(Get-Process -id $pid)
                      }
               $Objet2
          }|Wait-job|Receive-job
$Result.Nested.Level3[0][0][0]
```

```
16
```

On récupère bien les données des collections de l'objet personnalisé et de l'objet processus :

```
$Result.Process
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    340      30    49508      63848   614    0,62   4052 powershell
```

```
$Result.Process.StartInfo
```

```
System.Diagnostics.ProcessStartInfo
```

Par contre les données de l'objet imbriqué *StartInfo* sont absentes, à la place on reçoit le nom du type. La sérialisation est configurée dans un fichier de type *.ps1xml*, à partir de la version 3 on peut récupérer ces informations directement de la mémoire à l'aide du cmdlet **Get-TypeData** :

```
$Type=Get-TypeData -TypeName System.Diagnostics.Process
Type.SerializationDepth
```

```
0
```

Pour la classe *Process*, la profondeur de sérialisation est de zéro. Ce que signifie que les objets imbriqués ne sont pas inclus dans le traitement de sérialisation.

Cette configuration par défaut évite une surcharge réseau lors du transfert de l'objet sérialisé.

Pour la modifier, toujours sous la version 3, on utilise le cmdlet **Update-TypeData**. Ajoutez son appel dans l'exemple précédent :

```
$Result = Start-job {
            Update-TypeData -TypeName System.Diagnostics.Process `
             -SerializationDepth 2 -Force
            $Objet1=New-Object PSobject -property @{
…
```

Puis réexécutez le code :

```
$Result.Process.StartInfo
```

```
Verb                  :
Arguments             :
CreateNoWindow        : False
EnvironmentVariables  : {System.Collections.DictionaryEntry, System.Collections.DictionaryEntry,
```

Si vous avez compris le principe, vous savez ce qu'il vous reste à faire pour obtenir le détail de la propriété *EnvironmentVariables.*

Un objet désérialisé ne possède plus les méthodes de sa classe d'origine :

```
$Result.Process|gm -MemberType *method*
```

```
   TypeName : Deserialized.System.Diagnostics.Process
Name      MemberType   Definition
----      ----------   ----------
ToString  Method       string ToString(), string ToString(string format, ..
```

Ce qui provoquera ce type d'erreur :

```
$Result.Process.GetType()
```

```
Échec lors de l'appel de la méthode, car [Deserialized.System.Diagnostics.Process] ne contient pas de méthode appelée «GetType».
```

Voir aussi :

How objects are sent to and from remote sessions

http://blogs.msdn.com/b/powershell/archive/2007/05/01/object-serialization-directives.aspx

Une alternative pour Powershell version 2.

6.4 Gestion d'erreur

Ce chapitre suppose des connaissances de base sur la gestion des erreurs, si ce n'est pas votre cas vous pouvez consulter ce tutoriel.

6.4.1 Cas d'une erreur simple

Dans un job, les erreurs simples ne bloquent pas le déroulement du traitement :

```
$Error.Clear(); '1....5|Foreach {"Test"}' > C:\Temp\ScriptError.ps1
$Job=Start-Job -Name 'VisuEtat' -ScriptBlock {."C:\Temp\ScriptError.ps1"}`
|Wait-job
$Error.Count
```

```
0
```

```
$Job|Select-Object State,HasMoreData
```

```
State       HasMoreData
-----       -----------
Completed   True
```

```
$Job.Error
```

```
#vide
```

```
$Job.ChildJobs[0].Error
```

```
Au caractère C:\Temp\ScriptError.ps1:1 : 4
```

Résumons.

La collection $Error contenant toutes les erreurs est vide, l'état du job est '*Completed'*, la propriété *Error* du job ne contient aucune erreur, par contre celle du job enfant contient une erreur.

La récupération des données via **Receive-Job** émet une erreur et renseigne la collection $Error :

```
Receive-Job $job
```

```
Au caractère C:\Temp\ScriptError.ps1:1 : 4
```

```
$Error.Count
```

```
1
```

Le type de l'erreur reçue est *RemotingErrorRecord* :

```
$e=$Error[0]
$e.GetType().Fullname
```

```
System.Management.Automation.Runspaces.RemotingErrorRecord
```

L'instance comporte deux membres synthétiques, *writeErrorStream* qui permet d'utiliser la couleur associée aux erreurs lors de l'affichage de l'objet :

```
$e
```

```
Au caractère C:\Temp\ScriptError.ps1:1 : 4
```

Et *PSMessageDetails* pointant sur la propriété *InnerException*.

La classe contient une propriété *OrigineInfo* précisant l'origine de l'erreur :

```
$e.OriginInfo|fl
```

```
PSComputerName  : localhost
RunspaceID      : a252b779-030c-4995-990b-7913c231d820
InstanceID      : 00000000-0000-0000-0000-000000000000
```

Le type de l'exception est *RemoteException* :

```
$e.Exception.GetType().Fullname
```

```
System.Management.Automation.RemoteException
```

Et sa propriété *SerializedRemoteException* contient l'exception distante :

```
$e.Exception.SerializedRemoteException|gm
```

```
TypeName : Deserialized.System.Management.Automation.ParseException
```

L'erreur est bien une erreur de parsing du code.

Pour un appel à **Write-Error** nous aurions le type :

```
Deserialized.Microsoft.PowerShell.Commands.WriteErrorException
```

Dans notre exemple l'état du job est *'Completed'*, car l'erreur n'est pas bloquante :

```
$Job.JobStateInfo
```

```
State         Reason
-----         ------
Completed
```

La même propriété, mais sur le job enfant cette fois, contient la même information :

```
$Job.ChildJobs[0].JobStateInfo
```

```
State         Reason
-----         ------
Completed
```

Note sur la classe ***RemoteException :***

The remote instance of Windows PowerShell can be a separate application domain, process, or computer. This remote entity can be PS.exe (the default host application) or a custom host application.

The type of the original exception might not be available in the local application domain, process, or computer, so the original exception is available only as a serialized PSObject object.

6.4.2 Cas d'une erreur bloquante

Dans un job, une erreur bloquante arrête le déroulement du traitement :

```
@'
Throw "Déclenche une exception"
'@ > C:\Temp\ScriptError.ps1
$Job=Start-Job -Name 'VisuEtat' –ScriptBlock {. C:\Temp\ScriptError.ps1}
```

L'état du job est *'Failed'* :

```
$Job.JobStateInfo
```

```
State      Reason
-----      ------
Failed
```

Et c'est le job enfant qui détaille la raison de l'échec :

```
$Job.ChildJobs[0].JobStateInfo
```

```
State      Reason
-----      ------
Failed     System.Management.Automation.RemoteException: Déclenche …
```

Notez que la collection ***$Error*** n'est pas mise à jour tant que **Receive-Job** n'est pas appelé.

Cette propriété *Reason* contient l'exception déclenchée :

```
$Job.ChildJobs[0].JobStateInfo.Reason|gm
```

```
TypeName : System.Management.Automation.RemoteException
```

La collection *Error* du job est vide

```
$Job.Error
```

```
#vide
```

Et celle du job enfant également :

```
$Job.ChildJobs[0].Error
```

```
#vide
```

SAUF si le code exécuté émet des erreurs simples AVANT de déclencher l'erreur bloquante.

Pour un job à l'état *'Completed'* sa collection *Error* peut contenir des erreurs et pour un job à l'état *'Failed'* sa collection *Error* peut ne pas contenir d'erreur.

Nous pouvons en conclure que la gestion d'erreur d'un job doit tenir compte du contenu de ces deux propriétés, de plus l'appel à **Receive-Job** ne modifie pas ces informations.

Notez que la collection *Error* d'un job ne contient que les erreurs simples.

Le script *"\Sources\Scenario Erreur.ps1"* expose les différents cas.

6.4.3 Gestion de code retour d'un programme exécuté à distance

Dans ce scénario, disponible dans le script précédent, le job n'est pas en erreur et n'en renvoie aucune dans les flux d'erreur :

```
$Job=Invoke-Command -computername LocalHost  -scriptblock {
  $ErrorActionPreference="Continue"
  xcopy.exe Inconnu.txt Nouveau.txt 2>&1
} –AsJob |
 Wait-job|
 Get-job
$Error.clear()
$Result=Receive-Job $job
```

L'appel de **Receive-job** ne renseigne pas la collection $Error, mais renvoie dans le pipe un objet de type RecordError désérialisé et le message émis par **xcopy**. Les variables ***$?*** et ***$LASTEXITCODE*** du job distant ne sont pas propagées dans la session locale.

Si la variable ***$ErrorActionPreference*** est paramétrée avec "*Continue*", et puisque l'on redirige le flux d'erreur du programme externe vers la console, on doit analyser le résultat récupéré afin de déterminer si oui ou non l'appel à **xcopy** a déclenché une erreur.

```
$Result[0]
```

```
0 fichier(s) copi'(s)
```

```
$Result[1]
```

```
xcopy.exe : Fichier introuvable - Inconnu.txt
```

Remplaçons le contenu de la variable ***$ErrorActionPreference*** par "*Stop*"

```
$Job=Invoke-Command -computername LocalHost  -scriptblock {
  $ErrorActionPreference="Stop"
  xcopy.exe Inconnu.txt Nouveau.txt 2>&1
} –AsJob |
 Wait-job|
 Get-job
$Result=Receive-Job $job
```

Dans ce cas le job est en erreur et n'en renvoie aucune dans le flux d'erreur.

L'appel de **Receive-job** renseigne la collection $Error et la variable ***$Result*** est à *$null*.

L'erreur est du type *RemotingErrorRecord*, sa propriété Exception est du type *RemoteException*, et enfin sa propriété *SerializedRemoteException* :

```
$Result[0].Exception.SerializedRemoteException|gm
```

```
   TypeName : Deserialized.System.Management.Automation.RemoteException
```

On retrouve l'origine de l'erreur de la manière suivante :

```
$Error[0].Exception.SerializedRemoteInvocationInfo
```

```
MyCommand            : xcopy.exe
…
```

```
  TypeName: Deserialized.System.Management.Automation.ErrorRecord
```

Ici $Error[0] ne référence pas d'erreur liées à l'exécution du code sur le distant, seul *$Result* contient des erreurs désérialisées. Ce qui est un plutôt déroutant dans les premiers temps.

6.4.4 Intercepter les erreurs émises par Receive-Job

Lors de son appel **Receive-Job** renseigne la collection $Error et redéclenche en local les exceptions distantes, on peut les intercepter par un bloc try catch :

```
$Job=Invoke-Command -computername LocalHost -scriptblock {

  $ErrorActionPreference="Stop"
    write-output "Un"; write-output "Deux"

  xcopy.exe Inconnu.txt Nouveau.txt 2>&1

  write-output "Trois"
} -AsJob |
 Wait-job|
 Get-job
$Error.clear()

$ErrorActionPreference="Stop"
1..3| Foreach {
  try {
    $Result=Receive-Job $job
  } catch {
     $Error.Count
     Write-warning "Catch !"
  }
}
```

```
Un
Deux
2
AVERTISSEMENT : Catch !
2
AVERTISSEMENT : Catch !
2
AVERTISSEMENT : Catch !
```

Une fois les données du job récupérées et malgré l'absence du paramètre *–Keep* l'exception est redéclenchée à chaque appel de **Receive-Job**. Pour cet exemple la collection $Error est modifiée une seule fois.

6.5 RunspacePool

Si le critère de performance est important, les jobs locaux peuvent être remplacés par une mécanique basée sur des runspaces. Ce qui implique de recoder ce que PS propose nativement, à moins d'étudier et valider des solutions existantes.

Voir aussi :

http://www.nivot.org/post/2009/01/22/CTP3TheRunspaceFactoryAndPowerShellAccelerators

http://newsqlblog.com/2012/05/22/concurrency-in-powershell-multi-threading-with-runspaces/

Projets :

https://github.com/nightroman/SplitPipeline

https://poshrsjob.codeplex.com/

7 Suivi

Dans l'introduction j'ai parlé d'un traitement gérant d'autres traitements, sa tâche principale est de suivre le déroulement des jobs en cours afin de récupérer leurs données dès que possible, de libérer les ressources du job et enfin de démarrer les tâches en attente d'exécution.

La connaissance de [la gestion des événements sous Powershell] facilitera la lecture des prochaines pages. Le tutoriel est disponible dans les sources de ce tutoriel.

7.1 Changements d'état

Nous avons vu que le cmdlet *Wait-Job* suspend dans le même temps la gestion des événements déclaré *via* Powershell, il est probable que tous les cmdlets proposant un paramètre ***–Wait*** fonctionnent sur le même principe et posent le même problème.

Afin d'éviter l'usage du cmdlet *Wait-Job*, le suivi de l'exécution d'un job peut se faire à l'aide d'un événement déclenché lorsqu'il change d'état. Dans l'exemple suivant on ne gère que les états *Completed*, *Failed* et *Stopped* qui suffisent le plus souvent :

```
  #Job de traitement
$Job=Start-Job -Name 'VisuEtat' -ScriptBlock {Sleep -s 3;Get-Item C:\}

 #Job de surveillance du job de traitement
$null=Register-ObjectEvent $Job StateChanged -SourceIdentifier
"StateChanged_$($Job.Name)" -Action {
      $EventName=$EventArgs.JobStateInfo.ToString()
      $Msg="Etat précédent: $($EventArgs.PreviousJobStateInfo.State)"
      Write-Warning $Msg
      Switch ($EventName) {
      'Completed' {
                    $Msg="Job '$($Sender.Name)' dans l'état Completed."
                    Write-Warning $Msg
                    $global:datas =Receive-Job -id $Sender.Id -keep
                    Remove-job -id $Sender.Id
                    Break                   }
      'Failed'    { Write-Warning 'Job dans l''état Failed'; Break}
      'Stopped'   { Write-Warning 'Job dans l''état Stopped'; Break}
      default     {Write-Warning 'Cet état n''est pas géré : $EventName.'}
     }#Switch
   }  #Action
AVERTISSEMENT : Etat précédent: Running
AVERTISSEMENT : Job 'VisuEtat' dans l'état Completed.
$Datas
   Répertoire :…
```

Dans le bloc *'Completed'* du job d'événement, on récupère les données dans la variable *$global:Datas*, en étant de portée globale la console peut y accéder, puis on supprime le job 'VisuEtat'.

Si la gestion du résultat est simple, ajoutons la suppression de l'abonnement et de son job :

```
     }#Switch
       Write-Warning "Annule l'abonnement de l'event 'StateChanged'."
     UnRegister-Event -SubscriptionId $EventSubscriber.SubscriptionId
       Write-Warning "Supprime le job lié à la gestion de l'événement."
     Remove-job -id $EventSubscriber.Action.Id
    }  #Action
```

De cette manière ce code supprime tous les objets qu'il créé.

Cette mécanique est possible, car le code de gestion d'événement est hébergé dans un job situé dans le même état de session que la console et pas dans un autre processus Powershell, comme le fait **Start-Job**.

Les variables automatiques, telles que ***$EventSubscriber***, permettent au code du job d'accéder aux objets déclarés dans la console. Ainsi il peut se supprimer lui-même. En revanche la destruction se fait dans l'ordre inverse de la création.

Lors de l'annulation de l'abonnement son job associé est stoppé, ce qui rend possible sa suppression. Ne pas confondre ici l'objet job avec l'objet scriptblock de son paramètre *–Action*.

De prime abord cela semble complexe, alors qu'il ne s'agit que d'un assemblage de cmdlet et d'instructions s'imbriquant et d'un ensemble de comportements à connaitre. Avec la pratique ce code deviendra plus familier.

7.1.1 Utiliser une collection synchronisée

Dans le dernier exemple, le résultat du job surveillé est placé dans une variable globale. Si on utilise une seule variable pour plusieurs jobs, il est nécessaire d'utiliser une collection synchronisée afin de s'assurer qu'à un moment *T*, un seul job d'événement y ajoute des données :

```
$global:SynchronizedDatas=[System.Collections.ArrayList]::Synchronized(
                          (New-Object System.Collections.ArrayList(10) ))
```

Le code d'ajout devenant :

```
      Switch ($EventName) {
      'Completed' {
                    Write-Warning 'Job dans l''état Completed.';
                    $global:SynchronizedDatas +=Receive-Job -id $Sender.Id
                    Remove-job -id $Sender.Id
                    Break
                  }
…
```

7.2 Reste à faire

Pour faciliter le suivi, l'instruction *Switch* de l'exemple précédent devrait prendre en compte les états '*Failed'* et '*Stopped'*. Le décompte des tâches réalisées et celles en erreur déterminera le reste à faire.

Selon la volumétrie des informations de suivi, une base de données peut faciliter leur traitement.

Un exemple basé sur le cmdlet **Group-Object** :

```
Get-Job –Name LDJob*|Remove-Job –force
1..5|Foreach {Start-job -name "LDJob$_" {sleep 15;$Args[0]} -argumentlist
"Job$_" } > $null
6..10|Foreach {Start-job -name "LDJob$_" {sleep 10;$Args[0]} -argumentlist
"Job$_" } > $null
11..15|Foreach {Start-job -name "LDJob$_" {sleep 5;$Args[0]} -argumentlist
"Job$_" } > $null
  Do {
   Write-Warning "Wait"
   Start-Sleep -S 4
   $G=Get-Job –Name LDJob*| Group-Object -Property state
   $G|Format-Table Name,Count -AutoSize
  } until (($G[0].Name -eq 'Completed') -and  ($G[0].Count -eq 15))
```

```
Name        Count
----        -----
Running        13
Completed       2
AVERTISSEMENT : Wait
…
```

7.3 Rollback

Il n'existe pas de notion de rollback associé à un job, c'est à vous de le coder, sachez toutefois qu'un job de workflow peut implémenter des points de reprise. Souvenez-vous également que le provider Registry supporte les transactions.

7.4 Log

Il est préférable de gérer l'historique d'exécution dans un fichier texte, en cas de crash celui-ci persiste. L'usage d'outils spécialisés tels que *Log Parser* ou *Log Expert* permettront de suivre la progression de traitements ou d'effectuer des recherches.

Vous pouvez utiliser le module **Log4Posh** présent dans le répertoire des sources, vous trouverez sur cette page des explications sur son fonctionnement.

Le script suivant *'Sources\Log4Posh\Demos\DemoBackgroundJob.ps1'* configure un fichier de log pour le script principal et pour chaque job qu'il crée. Vous devez installer le module dans un des répertoires indiqué dans la variable d'environnement ***$PSModulePath***. Ce qui facilitera l'initialisation d'un job.

Ce mécanisme servira également à mémoriser les informations des possibles exceptions.

L'outil DebugView peut être utile lors de la mise au point, la fonction *'Sources\Tools\Write-Properties.ps1'* écrit sur un tel dispositif, sous réserve qu'on soit dans la même session Windows :

```
 !!! Modifier les chemins
&"C:\temp\Dbgview.exe";Start-Sleep 2
$Sb={.'\Sources\Tools\Write-Properties.ps1'}
Start-Job -InitializationScript $Sb  {
 Get-process -id $PID|
  Write-Properties -silently
}
```

7.5 Time out sur un job

Dans ce suivi, le fonctionnel peut nécessiter d'ajouter une durée limite d'exécution d'un job. Attention ce n'est pas du temps réel, ici tout ce que l'on sait, est que le traitement lié au timeout aura lieu si besoin.

Dans le cas où le traitement principal utilise des événements, l'usage du cmdlet **Start-Sleep** est exclu pour les raisons évoquées. Le traitement contenant déjà une gestion d'événements on peut baser cette fonctionnalité sur une seule instance de la classe Timer.

Le script de démo '*\Sources\Job-Timeout.ps1'* implémente cette fonctionnalité. Lors du déclenchement du timer, le code parcourt la liste des jobs concernés, ciblés par une norme de nommage, et calcule la durée d'exécution de chaque job *via* sa propriété *PSBeginTime* :

```
Get-Job -Name LD*|
 Where {($_.State -eq 'Running') -and (
        ([DateTime]::Now -$_.PSBeginTime) -gt $global:TimeOutJobInterval)
 }| Stop-Job # ou Remove-Job
```

Dans le cas où un job répond à la condition on change l'état du job et c'est le job du traitement de l'événement *StateChanged* qui prend le relais puisqu'on le lui a délégué !

La version 2 de Powershell ne propose pas de propriété *PSBeginTime*, une astuce est de placer cette information dans le nom du job. Consultez le script cité pour les détails d'implémentation.

7.5.1 PSBeginTime et PSEndTime

L'affectation d'une valeur à ces deux propriétés dépend de l'état du job.

Par défaut :

PSBeginTime : indique le début du job, valeur affectée lors de son démarrage.

PSEndTime : indique la fin du job, valeur affectée lors de son arrêt.

Pour un BackgroundJob elles sont toujours renseignées, que l'arrêt soit dû à une erreur ou forcé par le cmdlet **Stop-Job**. La propriété *PSBeginTime* du job enfant peut ne pas être renseignée si on l'arrête tout de suite après sa création (on ne lui laisse donc pas le temps de démarrer).

7.6 ThrottleLimit

Start-Job exécutant un job à la fois, il ne propose pas de paramètre limitant le nombre maximal de job exécuté en parallèle, le post suivant propose une solution basé sur une file d'attente synchronisée : Scaling and Queuing PowerShell Background Jobs

Le script "*\Sources\JobDemos\ThrottleLimit-Demo.ps1"* propose une autre démonstration de ce principe.

7.7 Événement réexpédié

Le forwarding peut être traduit par réexpédition, cette mécanique est prise en charge par WinRM. Elle permet d'informer une machine cliente qu'un événement a eu lieu sur une ou plusieurs machines distantes.

Cette possibilité peut également être utilisée à partir du processus d'un job, lui permettant d'émettre des événements vers son créateur qui est comme nous l'avons vu est dans un autre processus :

```
  #Déclare un event personnel et le traitement associé
Register-EngineEvent -SourceIdentifier PersonnalEvent -action {Write-
Warning "Réception d'un event émis d'un job"}

Start-Job -Name "Forward" -ScriptBlock {
     #Déclare un event personnel, le même que la session principale.
     #-Forward propage l'event entre deux process ou 2 machines
    Register-EngineEvent -SourceIdentifier PersonnalEvent -Forward


      #Crée un événement personnel
      #L'instruction précédente le propagera
    New-Event -SourceIdentifier PersonnalEvent -MessageData 10 > $null
 }
```

```
AVERTISSEMENT : Réception d'un event émis d'un job.
```

7.8 Wait-Event

Pour placer le script principal en attente sans bloquer la gestion d'événement, on s'appuie une fois de plus sur celle-ci, plus précisément sur une attente d'événement. Celle-ci se fait à l'aide du Cmdlet **Wait-Event**. Le script est bien attente et continue ainsi à recevoir les événements configurés.

Pour rappel, lors du déclenchement d'un évènement si l'abonnement associé (**Register-ObjectEvent**) déclare le paramètre *–Action* alors l'événement est consommé et **Wait-Event** ne le recevra pas puisqu'il est retiré de la file d'attente.

Pour que le cmdlet **Wait-Event** reçoive un événement on utilisera soit le cmdlet **Register-ObjectEvent** sans déclarer le paramètre *–Action*, soit le cmdlet **New-Event**.

Ceci dit, si notre script principal est en attente, il ne peut lui-même déclencher d'événement ☺

7.8.1 Créer un launcher

Pour générer des événements, on peut par exemple créer un eventlog particulier :

```
  #Nom de l'eventlog dédié hébergeant les demandes de job
$EventLogName='Launcher'

  #Nom d'une source pour l'eventlog dédié.
  #On y émet des demandes de job
$EventLogSourceName='ReceptionOrdre'

New-EventLog -logname $EventLogName -Source $EventLogSourceName
```

Puis dans une session Powershell dédiée, s'abonner à son événement EntryWritten :

```
  #Récupère l'eventlog à surveiller
$EventLog=Get-EventLog -List |Where {$_.log -eq $EventLogName}

$JobEventEntryWritten=Register-ObjectEvent $EventLog EntryWritten `
 -SourceIdentifier EventEntryWrittenInEventLog
```

Ensuite, n'importe quel processus peut y écrire une entrée normée, ici la propriété *eventID* est un numéro de commande à exécuter :

```
 $CmdTraitement=1
 $CmdInventaire=$CmdTraitement+1
 $CmdStopLauncher=$CmdTraitement+2

 Write-Eventlog -logname $EventLogName -source $EventLogSourceName `
 -eventID $CmdTraitement  -entrytype Information -message 'MesDonnées' `
 -category 1
```

Ce qui déclenche par rebond des événements dans la session Powershell en attente de les consommer :

```
 $CurrentEvent=Wait-Event

  #Ce switch permet de traiter des ordres et d'autres événements si besoin
 Switch ($CurrentEvent.SourceIdentifier) {

   "EventEntryWrittenInEventLog" {
     …
    switch ($Eventargs.Entry.EventID)
    {
       $CmdTraitement {
                          Start-Job { … } -Name 'CmdTraitement'
                          Break
                       }

       $CmdInventaire {
…
```

Par conséquent, la session Powershell utilisant **Wait-Event** termine son attente d'événement et lance un job selon votre paramétrage, puis le cycle d'attente recommence.

Le script *'\Sources\JobLauncher.ps1'* contient un exemple d'implémentation en deux parties.

8 Utiliser un Job distant

Il s'agit ici d'un job dont la demande est effectuée sur une machine locale et le traitement exécuté sur une ou plusieurs machines (1-to-many "fan-out"). Il va sans dire que le code exécuté est identique pour toutes.

WinRM doit être configuré sur la ou les machines utilisées, pour simplifier ce chapitre j'utilise uniquement des connexions sur *localhost* avec une configuration par défaut.

Pour la configuration de WinRM reportez-vous aux tutoriaux cités sur cette page.

En cas de problème, ceux-ci peuvent vous aider :

Troubleshooting WinRM and PowerShell Remoting
Collecting WinRM Traces

Voir également l'aide en ligne

```
Help about_Remote*
```

8.1 Exécution d'un RemoteJob

Pour démarrer un job distant :

```
$Job=Invoke-Command -ComputerName . -ScriptBlock { Get-Item c:\ } -AsJob
```

Si on reprend le script de surveillance de processus, on remarque que les processus créés différent de ceux liés à un BackgroundJob :

```
AVERTISSEMENT : Process créé C:\Windows\system32\wsmprovhost.exe\wsmprovhost.exe :
  C:\Windows\system32\wsmprovhost.exe-Embedding
AVERTISSEMENT : Process créé C:\Windows\system32\DllHost.exe\dllhost.exe :
  C:\Windows\system32\DllHost.exe /Processid:{E10F6C3A-F1AE-4ADC-AA9D-2FE65525666E}
# …
AVERTISSEMENT : Process détruit C:\Windows\system32\DllHost.exe\dllhost.exe
AVERTISSEMENT : Process détruit C:\Windows\system32\wsmprovhost.exe\wsmprovhost.exe
```

WSMProvHost.exe est lié au remoting WinRM et gère les accès à une session distante, le processus *DllHost.exe* est un mécanisme système de cloisonnement et me semble lié à **Invoke-Command**.

8.2 Invoke-Command

Le cmdlet **Invoke-Command** exécute un job dans une session distante temporaire, c'est à dire que son contexte d'exécution ne persiste pas entre deux appels. Si nécessaire il peut être associé à une session permanente, c'est alors le script appelant **Invoke-Command** qui décidera de la durée de vie de la ou des sessions distantes.

8.2.1 ThrottleLimit

Lors de l'appel à **Invoke-Command** l'usage de ce paramètre limite le nombre de job exécuté en parallèle. Tant que la limite définie n'est pas atteinte, ce cmdlet crée des jobs enfant dans l'état '*Running*', les jobs restant sont créés dans l'état *'NotStarted'*. Une fois qu'un des jobs démarré est terminé, un de ceux en attente démarre automatiquement.

Si la valeur limite précisée ou si la valeur par défaut (32) dépasse le seuil indiqué par l'item *WSMan::localhost\Shell\MaxProcessesPerShell*, qui est par défaut de 24, les jobs restant basculent dans l'état *'Failed'*.

Attention ce n'est pas une limite totale pour la session, mais une limite pour chaque job parent créé par **Invoke-Command.**

De plus :

- chaque nouveau job crée un processus *WSMProvHost.exe*, une fois le traitement du job terminé ce processus est détruit, les données renvoyées restent accessibles,
- les jobs enfants ne peuvent être supprimés indépendamment du parent,
- l'appel à *Receive-Job* est effectif pour tous les jobs de la collection ChildJobs, à moins de les cibler un par un.

Pour plusieurs centaines de machines un découpage par lots peut être envisagé.

8.3 Changements d'état

Ce type de job peut gérer plusieurs enfants, le suivi de leur exécution à l'aide d'événement se fait sur la collection *ChildJobs* et pas sur le job parent. Mais suite à quelques tests, la console principale se fige après le traitement de plusieurs changements d'état de jobs enfants. Ce n'est pas plus mal, car cette approche a pour inconvénient d'augmenter le nombre de job actifs.

Notez que la propriété *PSEndTime* du job parent est initialisée à *$Null* et est renseignée dès que le dernier job enfant est traité.

Par contre si un des jobs enfant reste dans l'état '*Blocked'*, cette propriété n'est pas mise à jour.

8.3.1 PSBeginTime et PSEndTime

L'affectation d'une valeur à ces deux propriétés dépend de l'état du job.

Par défaut :

PSBeginTime : indique le début du job, valeur affectée lors de son démarrage.

PSEndTime : indique la fin du job, valeur affectée lors de son arrêt.

Pour un RemoteJob elles ne sont pas toujours renseignées par exemple,

- les jobs enfants à l'état *'NotStarted'* n'ont aucune des deux de renseignée, car ils sont en attente,
- si on appel **Stop-Job** sur un job enfant à l'état *'NotStarted'*, alors la propriété *PSEndTime* est renseignée, mais pas *PSBeginTime* car il n'a jamais démarré,

- idem si le nom d'ordinateur sur lequel doit s'exécuter le traitement n'est pas joignable, l'état du job enfant sera alors *'Failed'*,
- évidement les jobs enfants à l'état *'Running'* n'ont pas encore leur propriété *PSEndTime* de renseignée.

8.3.2 Wait-Job

Si on utilise **Wait-Job** sans paramètre sur le job parent :

```
$Job=Invoke-Command -ComputerName 'localhost','localhost' -JobName Parent
-AsJob -Scriptblock { Sleep -s 5; Get-Item C:\}


$Job|Wait-Job
```

Le cmdlet attend que l'exécution de tous ses enfants soit terminée puis renvoie le job parent en tant que résultat.

Si on déplace l'attente sur les jobs enfants :

```
Wait-Job $job.ChildJobs
```

Le cmdlet attend la fin d'exécution de tous les jobs enfants et les renvoie en tant que résultat.

Pour récupérer un job enfant dès qu'il est terminé ceci renverra toujours le premier job terminé :

```
Wait-Job $job.ChildJobs -Any
```

Ce qui est d'un intérêt limité. De plus le paramètre -*State* ne cible que les jobs parent, car on ne peut écrire :

```
Wait-Job –Job $job.ChildJobs –Any –State Running
```

Dans le cas où l'on souhaite traiter les jobs enfants au fur et à mesure, si un des jobs enfant est dans l'état *Blocked* ou s'il reste dans l'état *Running* indéfiniment alors **Wait-Job** restera en attente. Pour éviter ce scénario une solution est de gérer un temps d'attente maximum sur chaque job enfant.

Selon la valeur de *ThrottleLimit* et du nombre de jobs enfants, certains peuvent se trouver dans l'état *NotStarted*, on doit donc les filtrer.

Le script *"\Sources\JobDemos\SuiviJobEnfants.ps1"* est une possible implémentation de cette gestion. Je n'ai pas pris en compte le suivi et la récupération des résultats propre à chaque traitement et une gestion d'erreurs, comme ceux dus aux quotas, reste à implémenter.

Note :

Chaque job possède une propriété *Finished* permettant une autre approche d'attente :

```
$null = $job.ChildJobs[1].Finished.WaitOne() #timeout Possible
$null = [System.Threading.WaitHandle]::WaitAll(@($job1.Finished,
$job2.Finished))
```

8.3.3 Receive-Job

Chaque résultat reçu possède une propriété *PSComputerName* contenant le nom de la machine distante. La propriété *Location* du job parent contient le nom de la machine locale.

Vous trouverez ici un exemple de récupération du résultat des jobs enfants.

8.4 Gestion d'erreur

Les jobs distants sont exécutés sur des machines dont la version de l'OS, Powershell ou du processeur peuvent être différentes de la machine locale. Votre code et vos tests doivent en tenir compte.

En cas d'erreur dans un job enfant, l'état du parent change, mais les autres jobs en cours continuent leur exécution et ceux en attente le restent.

Lors de vos tests si vous forcez la suppression de ces jobs, vous pouvez vous retrouver avec des instances orphelines du processus *wsmprovhost.exe*.

Si vous réitérez cette opération le nombre maximal de ce processus peut être atteint, dans ce cas stoppez le service WinRM puis relancez-le.

Note :

La configuration des sessions de remoting influencera le comportement de vos jobs distants : occupation mémoire, temps de traitements, etc.

```
Dir WSMan:\localhost -rec|
 Where Name -match 'Max'|
 Sort Name|Select Name,Value
```

```
Name                                Value
----                                -----
MaxBatchItems                       32000
MaxConcurrentCommandsPerShell       1000
MaxConcurrentOperations             1500
MaxConcurrentOperations             4294967295
…
```

```
Dir WSMan:\localhost -rec|
 Where Name -match 'Max'|
 Sort Name|
 Group psParentpath|
 Select name
```

```
Name
----
Microsoft.WSMan.Management\WSMan::localhost
Microsoft.WSMan.Management\WSMan::localhost\Plugin\microsoft.powershell32\Quotas
Microsoft.WSMan.Management\WSMan::localhost\Plugin\microsoft.powershell\Quotas
Microsoft.WSMan.Management\WSMan::localhost\Plugin\microsoft.powershell.workflow\Quotas
…
```

Voir aussi : Breaking change

9 Le paramètre -AsJob

Son usage permet d'exécuter le traitement d'un cmdlet en tâche de fond sans utiliser **Start-Job.**

Voici les cmdlets qui le proposent sous Powershell version 3 :

```
gcm -ParameterName AsJob –all -CommandType cmdlet
```

CommandType	Name	ModuleName
Cmdlet	Get-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Invoke-Command	Microsoft.PowerShell.Core
Cmdlet	Invoke-WmiMethod	Microsoft.PowerShell.Management
Cmdlet	Remove-WmiObject	Microsoft.PowerShell.Management
Cmdlet	Restart-Computer	Microsoft.PowerShell.Management
Cmdlet	Set-WmiInstance	Microsoft.PowerShell.Management
Cmdlet	Stop-Computer	Microsoft.PowerShell.Management
Cmdlet	Test-Connection	Microsoft.PowerShell.Management
Workflow	Invoke-AsWorkflow	PSWorkflowUtility

Ceux-ci possèdent également le paramètre *ComputerName*.

9.1 Implémenter AsJob

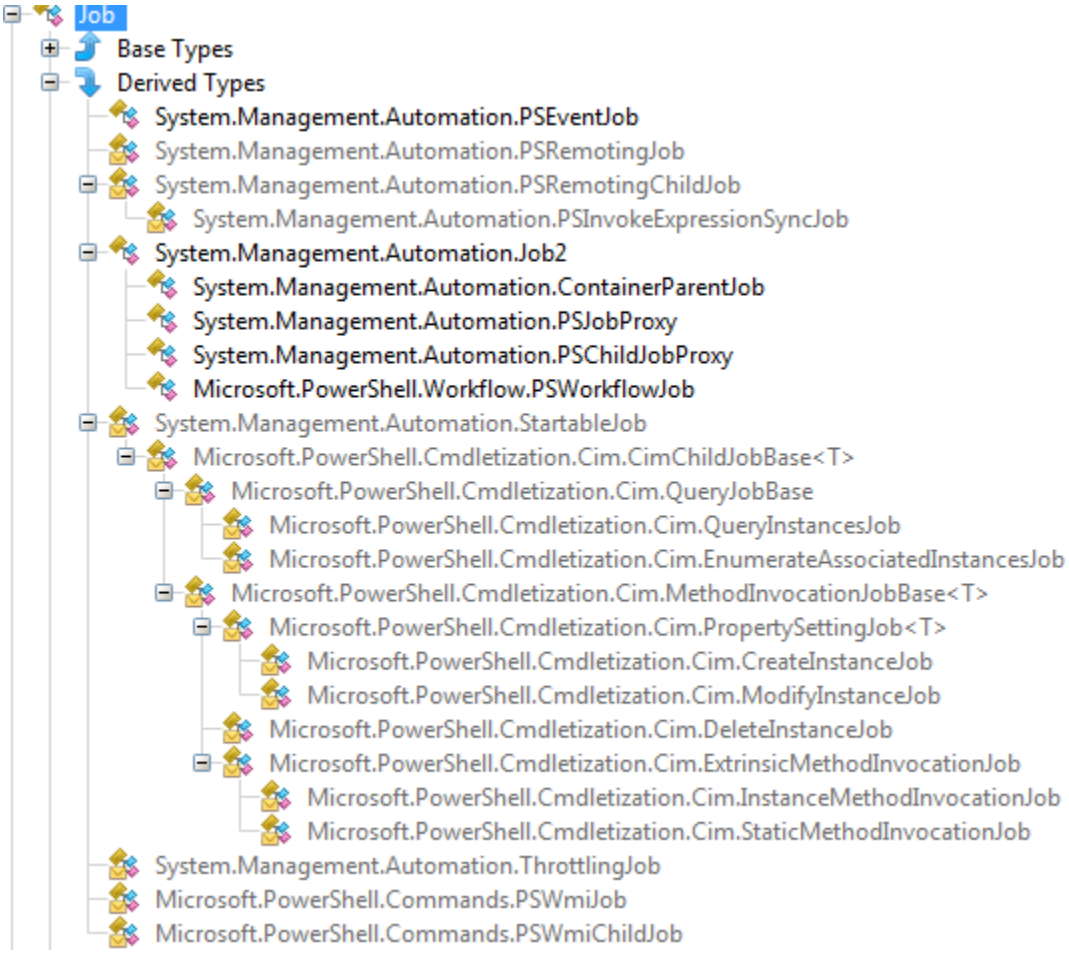
Le script de démonstration *'\Sources\AsJobFunction.ps1'* propose une implémentation dans une fonction et *'\Sources\AsJobScript.ps1'* dans un script.

Pour un cmdlet C# la page suivante *how-to support job* explique les étapes à suivre.

10 Types de job

La version 2 propose les classes suivantes, notez que les classes grisées ne sont pas publiques :

La version 3 propose celles-ci :



Comme indiqué dans la documentation MSDN :

« Le support de l'asynchrone se fait au travers de deux modèle (pattern) :

1. *Un modèle classique (Begin and End).*
2. *Un modèle basé sur les événements.*

Contrairement à l'API PowerShell, qui utilise le modèle classique, l'API des Job utilise le modèle basé sur les événements. »

Chaque job parent possède une propriété nommée *PSJobTypeName*, si le contenu est renseigné il indique un nom de type qui peut ne pas être identique au nom de la classe du job. Cette propriété permet par exemple de filtrer la collection de job existant dans la session :

BackgroundJob	Job local.
RemoteJob	Job distant.
PSWorkflowJob	Job de workflow.
PSEventJob	job d'événement.
ScheduledJob	Job planifié.
WmiJob	Job d'interrogation d'un référentiel WMI. Interroge un référentiel via le protocole DCOM.
CimJob	Job de manipulation de classe CIM. Cf. CDXML Interroge un référentiel via le protocole WsMan (compatible avec WMI).
Custom job	Job personnalisé. Les types *PSWorkflowJob* et *ScheduledJob* peuvent être considéré ainsi.

On peut également consulter sa propriété *PsTypeNames* pour connaitre ses classes ancêtres.

10.1.1 PSWorkflowJob

Les workflows Powershell s'appuient sur le framework Windows Workflow Foundation.

Rapidement, on peut suspendre l'exécution d'un job, puis la reprendre, il peut lui-même se suspendre et être persistant, par exemple lors d'un traitement local redémarrant la machine.

10.1.2 PSEventJob (job d'événement)

Ce type de job est traité dans le chapitre 5 *'PowerShell délègue un job'* du tutoriel sur les événements précédemment cité.

10.1.3 ScheduledJob

Le module *PSScheduledJob* ajoute les cmdlets suivants dédiés à la gestion de tâches planifiées :

Enable-ScheduledJob
Disable-ScheduledJob
Get-ScheduledJob
Set-ScheduledJob
Register-ScheduledJob
Unregister-ScheduledJob

New-ScheduledJobOption
Get-ScheduledJobOption
Set-ScheduledJobOption

Ainsi que ceux liés aux *JobTrigger* qui ne sont pas des jobs, mais des déclencheurs de jobs planifié :

Add-JobTrigger
Disable-JobTrigger
Enable-JobTrigger
Get-JobTrigger
New-JobTrigger
Remove-JobTrigger
Set-JobTrigger

Les *ScheduledJobs* sont enregistrés, au format XML, dans le répertoire :

$Home\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs

Consultez les fichiers d'aide pour plus de détails :

```
Help about_Scheduled*
```

10.1.4 Wmi & Cim

Le cmdlet **Register-WMIEvent** crée un job d'événement WMI, sa propriété *PSJobTypeName* n'est pas renseignée, sa collection *ChildJobs* est vide et par défaut son état est *NotStarted*.

Pour **Get-WmiObject** Powershell se connecte au référentiel WMI de l'ordinateur indiqué et ne crée pas de processus enfant :

```
Get-WmiObject Win32_Processor -asJob
```

```
Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
32     Job32           WmiJob          Running       True            localhost            Get-WmiObject Win32_Pr...
```

Le paramètre –*ComputerName* permet d'indiquer une liste de servers.

Le script *'Sources\cdxml\Get-DataProcess.ps1'* crée un job de type CimJob, la classe qui l'implémente, *System.Management.Automation.ThrottlingJob,* est interne au runtime.

Les cmdlets créés par ce type de module disposent automatiquement des paramètres suivants :

–AsJob, – ThrottleLimit et *–CimSession*

10.1.5 Custom job

L'ajout de la classe **Job2** dans le runtime Powershell permet dorénavant de créer ses propres Job, ceux-ci s'intégrant dans la mécanique de base de gestion des jobs.

Le projet ***JobSourceAdapter*** est une démo issue du SDK Powershell, il crée un cmdlet dont le rôle est de surveiller les modifications d'un fichier afin de le recopier dans un autre répertoire.

Le script *'\Sources\C#\ClassePersonnelleDeJob.ps1'* propose une démonstration de l'usage de ce type de job.

11 Liens

Powershell version 5 :

http://trevorsullivan.net/2014/10/04/powershell-5-0-background-jobs/

http://www.youtube.com/watch?v=EACmPHhNu3Y

Many-to-1 "fan-in" - delegating administration:
Delegating administration of a server by hosting PowerShell inside the service.

Aleksandar Nikolic – Delegation with Remoting.

http://www.manning.com/payette2/SampleCh-13.pdf

12 Conclusion

La création et l'usage de job est simple, l'aide proposée sur les cmdlets de base est suffisante pour un débutant. La simplification de la gestion de threads est remarquable, une fois encore l'équipe de développement a fait que ce shell rend service sans avoir à s'occuper de l'intendance technique.

Par contre dès que l'on veut implémenter une solution plus élaborée, les connaissances prérequises sont tout de suite plus nombreuses. Je trouve que la documentation des états des jobs est sommaire on se retrouver donc à progresser à tâtons.

La plupart des scénarios documentés que l'on trouve sur le net ne référencent que le scénario *des beaux jours* ciblant quelques machines, la réalité me semble tout autre.

On lit souvent sur le sujet des jobs sous Powershell qu'il est possible de traiter plusieurs centaines, voire milliers de machines via des jobs, je n'en doute pas. Le plus intéressant eut été de nous le démontrer.

Prêt pour une démo ?