Writing a Custom Server Agent

Technical Note

VERSION: 1.2

UPDATED: August 2019

Contents

Writing a Custom Server Agent

This document Shows you how to create your own load balancing server agent to use in conjunction with the **resource based (adaptive)** Virtual Service (VS) scheduling method, one of the VS **Scheduling Method** choices in the VS **Standard Options**. We'll focus primarily on the requirements for creating and configuring such agents on arbitrary server operating systems (for example, the protocol used by the agent to communicate load values to LoadMaster), and less on the mechanics of generating a load value using specific OS utilities. Nevertheless, we'll present simple examples of implementing and deploying a server agent on both a Windows and Linux system.

In the remainder of this document, we'll refer to the **resource based (adaptive) scheduling method** simply as **adaptive scheduling**.

Customer Server Agents and Adaptive Scheduling

Adaptive scheduling requires the following *on every RS in the VS*:

1. There is an HTTP server running on the RS and the endpoint for this HTTPS server is accessible to the LoadMaster.
2. There is a server agent installed and running on the RS that does the following:
    a. Periodically queries the RS operating system for whatever system load statistics are available and desired.
    b. Uses the load values obtained from the operating system to arrive at an integer value.
    c. Places the integer value in a file underneath the root directory of the HTTP server running on the RS. This is the file that the LoadMaster will request using an HTTP GET command. *It must contain a single integer value and nothing else*.

The server agent and an appropriately configured HTTP server must be running on all RSs in a VS *before* selecting adaptive scheduling on the VS.

On the LoadMaster side:

1. Once adaptive scheduling is selected on a VS, LoadMaster periodically attempts to retrieve from each RS the file populated by the server agent, with the integer load value.
    - Until LoadMaster successfully retrieves an integer load value from a Real Server, it will be considered *unavailable*.
2. Once an integer load value is retrieved, LoadMaster modifies the Real Server's status by interpreting the integer retrieved according to the table below.

Agent Response	Action Taken	Description
No HTTP Response	**Dynamic Weight = 0**	If LoadMaster gets *no response* when it attempts to get the file containing the load value, then the Real Server is assumed to be unavailable and it is marked down by setting the server's dynamic weight to 0, so that no new connections are sent to it. Currently open connections are not affected. This typically happens when either the Real Server is unavailable, or the required HTTP server is not responding to the LoadMaster's query.
File does not exist	**Dynamic Weight = 100**	If the HTTP server is running on the Real Server and responds to LoadMaster's query, it may send a response that indicates the file requested does not exist. In this case, the server is assumed to be available, it is marked up, and the server dynamic weight is set to 100.
File is empty	**Dynamic Weight = 100**	If the HTTP server is running on the Real Server and responds to LoadMaster's query with an empty file (the file contains no data at all), the server is assumed to be available, it is marked up, and the server dynamic weight is set to 100.
File contains a non-integer value	**Dynamic Weight = 100**	If the file returned by the HTTP server running on the Real Server contains a non-integer value, then the server is assumed to be available, it is marked up, and the server dynamic weight is set to 100.
< 0	**Dynamic Weight = 100**	If the file returned by the HTTP server running on the Real Server contains a negative integer value (e.g., -1), then the server is assumed to be available, it is marked up, and the **server dynamic weight is set to 100**.
0	**Switch to the round robin scheduling method**	If the file returned by the HTTP server running on the Real Server contains an integer value of 0 (zero), this forces LoadMaster to temporarily fall back to using the round robin scheduling method *until the server agent returns a value other than 0* on *a subsequent query*. This return value is intended to be used in a situation where the server agent code cannot for some reason arrive at a valid integer value to return to LoadMaster. This might happen if, for example, the server is not responding to the agent's queries for load values.

Agent Response	Action Taken	Description
1-100	**Dynamic Weight set appropriately**	If the file returned by the HTTP server running on the Real Server contains an integer value between 1 and 100, this is interpreted as a percentage of fully loaded (1=lowest load, 100 = fully loaded). LoadMaster sets the dynamic weight appropriately: the higher the number returned by the server agent, the lower the dynamic weight of the server will be set.
101	**Dynamic Weight = 0**	If the file returned by the HTTP server running on the Real Server contains an integer value of 101, the server is considered to have reached its maximum load and it is marked down. Server dynamic weight is set to 0, so that no new connections are sent to it. Currently open connections are not affected.
102	**Dynamic Weight = 0 plus optional connection drop**	If the file returned by the HTTP server running on the Real Server contains an integer value of 102, this means essentially the same as returning 101, except that LoadMaster will also drop all currently open connections if the **Drop connections on RS failure** option is set on the associated VS.
> 102	**Dynamic Weight = 0**	A return of 103 or greater is treated the same as a return of 101.

On any RS operating system, a straightforward manner in which to implement a server agent is to write a shell script or executable program that runs peridoically via a standard system scheduler, like `cron` on Linux systems or the Task Scheduler on Windows systems. This script would use standard utilities to generate the load value and place it in a file under the HTTP server root.

Such a script could be as simple as this Linux *bash* script.:

```
#! /bin/bash
top –l1 | awk '/CPU Usage/ {printf "%d\n", $7} > /path/load
```

The above uses the `top` command and a short `awk` script to copy the current CPU load value in a file – the *path* above specifies the path to the root of an HTTP server on the real server. The file name 'load' is the default file name retrieved by LoadMaster. [This and other adaptive load balancing options are located in the UI under **Rules & checking > Check Parameters > Adaptive Parameters**.]

Alternatively, you could instead write a CGI program or any other program that the HTTP server can be configured to run when LoadMaster sends a GET for the required file to the HTTP server. This program would generate the integer load value when LoadMaster requests the file from the server and the HTTP server would return it to LoadMaster.

There are advantages and disadvantages to either approach. For example:

- Using a static file populated via a `cron` job is a simple and easy-to-maintain option, but also means that the data in the file could be stale when it's accessed by LoadMaster. The interval at which the load value is generated should be based on the known performance profile of the server.
- Using a program that runs each time LoadMaster checks performance means that you'll get the most accurate data, but the server agent itself could contribute to system load if CPU, Disk I/O, and network performance were all being continually checked by the server agent.

The approach you choose will most likely depend on the needs of your configuration and the tools you typically use.

In terms of internal processing, a server agent can be as complicated as you like, depending on your application. If your application is completely CPU-intensive, then a simple script like the two-line `bash` script above may be enough. But, if your application also generates significant disk I/O or network-traffic, you'll probably want to consider the load on those resources as well as CPU usage to get a more detailed view of server performance.

Sample Linux Agent Script

The following script is an example of gathering load information on CPU and Memory, and then combining them into a single integer value that LoadMaster can use for adaptive scheduling. It's written as a `bash` shell script and is intended to be run on a Linux server via a privileged `crontab` file. See the Linux manual pages for `bash`, `crontab`, and the commands used in the script to gather the required statistics.

This script is intended solely as an example, although it could be used in a demonstration or proof of concept of how to configure adaptive scheduling.

```
#! /bin/bash
#
# Very simple adaptive server agent program, intended to be run via a root
# or similarly privileged crontab(1) entry to periodically populate
# the file LoadMaster expects to be available via HTTP on the Real Server.

#### DEFAULTS

# This is the default filename expected by LoadMaster. This setting must
# match what's on "Rules & Checking > Adaptive Parameters" in the LM WUI.

LOAD_FILE_PATH=/load

# This is the default doc root of apache2 on an ubuntu server, and where
# we'll place the 'load' file defined above for LoadMaster to GET.

DOCROOT=/var/www/html
```

```
# Define the weights to be used for CPU and Memory.
# Change these numbers to indicate the percentage weight that each
# statistic should carry when figuring the overall load value to
# return to LoadMaster. By default, these are set to 0.25 for CPU and 0.75
# for Memory, so the effect of the weighting is apparent. Weights must
# add up to 1.

CPUweight=0.25
MEMweight=0.75

#### GET MEMORY STATS

# Get memory usage from the free(1) command using a simple awk command and
# calculate the percentage of memory used. Since the shell does integer
# arithmetic only, we'll use bc(1) to do the math.

MEMtotal=`free | awk '/Mem:/ {print $2}'`
MEMstatus=$?
MEMused=`free | awk '/Mem:/ {print $3}'`
MEMload=`echo "scale=3;( $MEMused / $MEMtotal ) * 100" | bc`
# Round the answer to the nearest integer
MEMloadrnd=`echo "$MEMload + 0.5" | bc`
MEMloadrnd=`echo "scale=0; $MEMloadrnd/1" | bc`

#### GET CPU LOAD
# Note: Some version of top(1) report avg. CPU usage since time of last
# reboot in the first iteration, so we'll take the second iteration value.

CPUidletop=`top -b -n2 -d.2 | awk '/%Cpu/ {print $8}'`
CPUidle=`echo $CPUidletop | cut -d" " -f2`
CPUload=`echo "scale=3;100 - $CPUidle" | bc`
CPUstatus=$?
# Round the answer to the nearest integer
CPUloadrnd=`echo "$CPUload + 0.5" | bc`
CPUloadrnd=`echo "scale=0; $CPUloadrnd/1" | bc`

#### CHECK IF STATS ARE A VALID % (between 1 and 100)
# VALID_STATS=1 means stats are valid.
# if stats are not valid, we want to return 0 to LM to disable adaptive
# LB and use round robin until a different value is received.

VALID_STATS=1
if [ $MEMloadrnd -lt 1 -o $MEMloadrnd -gt 100 -o $CPUloadrnd -lt 1 \
   -o $CPUloadrnd -gt 100 ]
then
   VALID_STATS=0
   RETURNED_LOAD=0
fi

#### CALCULATE THE WEGHTED AVERAGE OF $CPUload AND $MEMload.
# Only do this if stats were valid - i.e., $VALID_STATS=1.
```

```
if [ $VALID_STATS -eq 1 ]
then
   RSload=`echo "( $CPUweight * $CPUloadrnd ) + \
      ( $MEMweight * $MEMloadrnd )" | bc`
   RSloadrnd=`echo "$RSload + 0.5" | bc`
   RSloadrnd=`echo "scale=0; $RSloadrnd/1" | bc`
   RETURNED_LOAD=$RSloadrnd
fi

#### DETERMINE IF WE SHOULD RETURN 101 OR 102.
# Returning 101 or 102 in the case of a highly loaded system is somewhat
# a matter of policy. In this example, we'll return 101 (take the server
# out of rotation) when stats are valid AND the aggregated load value
# is above 90%, to allow more resources to become available. If the
# aggregate load goes above 96%, we'll return 102 (which takes the server
# out of rotation AND drops all current connections). Presumably, once
# more resources are available, load values will decrease, and the
# server agent will return a lower value on a subsequent run.

if [ $VALID_STATS -eq 1 -a $RSloadrnd -gt 90 ]
then
   RETURNED_LOAD=101
elif [ $VALID_STATS -eq 1 -a $RSloadrnd -gt 96 ]
then
   RETURNED_LOAD=102
fi

#### PUT THE LOAD VALUE IN THE FILE.
# If stats were not valid above, then return 0 -- this disables adaptive
# scheduling and uses round robin instead; else, return the computed load.

if [ $VALID_STATS -eq 0 ]
then
   RETURNED_LOAD=0
fi
echo $RETURNED_LOAD > ${DOCROOT}${LOAD_FILE_PATH}

#### PRINT EVERYTHING TO STDOUT FOR DEBUGGING
# This allows for running the script from the command line to test in
# your environment -- or, if the script is run via cron, cron will
# capture the output and email it to you so you can debug any issues.
# Once you're happy the script is working properly, comment these lines
# out to avoid being emailed output by cron every time the script runs.

echo RETURNED_LOAD = $RETURNED_LOAD
echo RSload = $RSload
echo RSloadrnd = $RSloadrnd
echo MEMweight = $MEMweight
echo MEMtotal = $MEMtotal
echo MEMused = $MEMused
```

```
echo MEMload = $MEMload
echo MEMloadrnd = $MEMloadrnd
echo MEMstatus = $MEMstatus
echo CPUweight = $CPUweight
echo CPUidle = $CPUidle
echo CPUload = $CPUload
echo CPUloadrnd = $CPUloadrnd
echo CPUstatus = $CPUstatus

#### EXIT

if [ $VALID_STATS -eq 1 -a $MEMstatus -eq 0 -a $CPUstatus -eq 0 ]
then
# success
   exit 0
else
# error
   exit 1
fi

#### EOF
```

Sample Windows Agent Script

The following script is similar in function to the Linux script presented in the previous section, but is written in Windows PowerShell rather than the Linux *bash* shell and uses WMI () commands to do the job of gathering performance data. The script can be set to run periodically as a Windows Task and assumes that there is an already running IIS server on the Windows Real Server that will respond to the LoadMaster's GET request for the load value file created by the server agent.

This script is intended solely as an example, although it could be used in a demonstration or proof of concept of how to configure adaptive scheduling.

```
#
# Very simple adaptive server agent program, intended to be run via a
# Task Scheduler Task with system level privileges, to periodically
# populate the file LoadMaster expects to be available via HTTP on the
# Real Server.

#### DEFAULTS

# This is the default filename expected by LoadMaster. This setting must
# match what's on "Rules & Checking > Adaptive Parameters" in the LM WUI.

$LOAD_FILE_PATH="\load"

# This is the default root of a Windows IIS server. Change this to match
# the location where your HTTP server will require the load file (above)
# to be placed so that LoadMaster can access it using an HTTP GET request.
```

```
$DOCROOT="\inetpub\wwwroot"

# Define the weights to be used for CPU and Memory.
# Change these numbers to indicate the percentage weight that each
# statistic should carry when figuring the overall load value to
# return to LoadMaster. By default, these are set to 0.25 for CPU and 0.75
# for Memory, so the effect of the weighting is apparent. Weights must
# add up to 1.

$CPUweight=0.25
$MEMweight=0.75

#### GET MEMORY STATS

[int]$MEMtotal=(Get-WmiObject -Class
         win32_operatingsystem).TotalVisibleMemorySize
[int]$MEMfree=(Get-WmiObject -Class
         win32_operatingsystem).FreePhysicalMemory
[int]$MEMused=$MEMtotal - $MEMfree
$MEMloadpct=($MEMused / $MEMtotal).tostring("P")
[int]$MEMload=($MEMused / $MEMtotal * 100)

#### GET CPU LOAD

$CPUload=Get-WmiObject win32_processor | select -Expand LoadPercentage

$VALID_STATS=1
$RETURNED_LOAD=1
if ( $MEMload -lt 1 -or $MEMload -gt 100 -or $CPUload -lt 1 -or $CPUload -
         gt 100 )
{
   $VALID_STATS=0
   $RETURNED_LOAD=0
}

#### CALCULATE THE WEGHTED AVERAGE OF $CPUload AND $MEMload.
# Only do this if stats were valid - i.e., $VALID_STATS=1.

if ( $VALID_STATS -eq 1 )
{
   [int]$RSload=( $CPUweight * $CPUload ) + ( $MEMweight * $MEMload )
   $RETURNED_LOAD=$RSload
}

#### DETERMINE IF WE SHOULD RETURN 101 OR 102.
# Returning 101 or 102 in the case of a highly loaded system is somewhat
# a matter of policy. In this example, we'll return 101 (take the server
# out of rotation) when stats are valid AND the aggregated load value
# is above 90%, to allow more resources to become available. If the
# aggregate load goes above 96%, we'll return 102 (which takes the server
```

```
# out of rotation AND drops all current connections). Presumably, once
# more resources are available, load values will decrease, and the
# server agent will return a lower value on a subsequent run.

if ( $VALID_STATS -eq 1 -and $RSload -gt 90 )
{
   $RETURNED_LOAD=101
}
elseif ( $VALID_STATS -eq 1 -and $RSload -gt 96 )
{
   $RETURNED_LOAD=102
}

#### PUT THE LOAD VALUE IN THE FILE.
# If stats were not valid above, then return 0 -- this disables adaptive
# scheduling and uses round robin instead; else, return the rounded load.

if ( $VALID_STATS -eq 0 )
{
   RETURNED_LOAD=0
}
echo $RETURNED_LOAD > $DOCROOT$LOAD_FILE_PATH

#### PRINT EVERYTHING TO STDOUT FOR DEBUGGING
# This allows for running the script from the command line to test in
# your environment. Once you're happy the script is working properly,
# comment these lines out.

echo "VALID_STATS = $VALID_STATS"
echo "RETURNED_LOAD = $RETURNED_LOAD"
echo "MEMload = $MEMload"
echo "MEMloadpct = $MEMloadpct"
echo "CPUload = $CPUload"

#### EXIT with appropriate status
if ( $VALID_STATS -eq 1 )
{
# success
   exit 0
}
else
{
# invalid stats -- error
   exit 1
}

#### EOF
```

Last Update

This document was last updated in August 2019.