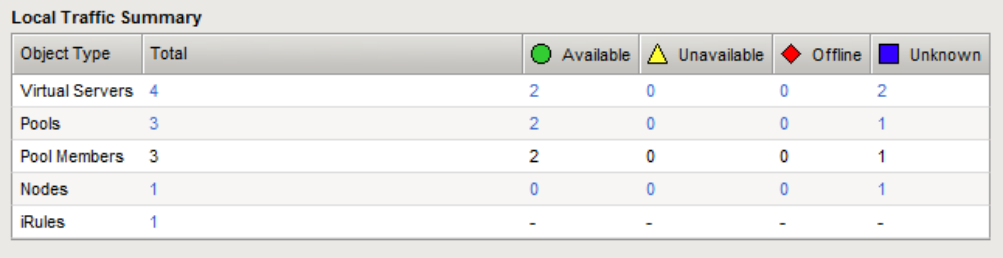
# iControl Apps - #02 - Local Traffic Summary

**Joe Pruitt, 2008-26-06**

The BIG-IP Management GUI has a feature called the Network Map that includes a summary of the objects on the system and their current statuses. This tech tip will take that GUI component and break it down into the underlying iControl method calls and present an alternative implementation from within Windows PowerShell.

### Introduction

The Network Map component of the BIG-IP Management GUI has two components. The first component is a Local Traffic Summary table that lists the Virtual Servers, Pools, Pool Members, Nodes, and iRules with their respective statuses. An example of this table can be seen in the image below:



**Local Traffic Summary**

| Object Type | Total | Available | Unavailable | Offline | Unknown |
|---|---|---|---|---|---|
| Virtual Servers | 4 | 2 | 0 | 0 | 2 |
| Pools | 3 | 2 | 0 | 0 | 1 |
| Pool Members | 3 | 2 | 0 | 0 | 1 |
| Nodes | 1 | 0 | 0 | 0 | 1 |
| iRules | 1 | - | - | - | - |

This application will replicate that table and show you how to use client side caching techniques to make the application run faster. As I mentioned above, we will be using Windows PowerShell with the iControl CmdLets for PowerShell available from the PowerShell DevCentral Labs website. The code listed below can be consolidated into a single script file (LocalTrafficSummary.ps1) to be run directly from the command line.

First, I'm going to define the parameters for the application. This can be done with the param statement. We'll be needing the BIG-IP address, username, and password. I'll also include here the declarations for a couple of global variables we'll be using for caching.

```
param (
  $g_bigip = $null,
  $g_uid = $null,
  $g_pwd = $null
)

# Set strict level debugging.
Set-PSDebug -strict;

# A few global variables to cache to avoid multiple duplicate iControl requests.
$g_vs_list = $null;
$g_vs_pool_list = $null;
$g_vs_rule_list = $null;
```

### Initialization

The entry code checks to see if the parameters were passed in correctly. If not, a usage message will be displayed. If the parameters were passed in correctly, it will attempt to initialize the iControl SnapIn and authenticate with the BIG-IP and then make a call to the local Print-LocalTrafficSummary function.

```
#--------------------------------------------------------------------------
# function Show-Usage
#--------------------------------------------------------------------------
```

```
function Show-Usage()
{
  Write-Host "Usage: LocalTrafficSummary.ps1 host uid pwd";
  exit;
}
#-------------------------------------------------------------------------
#
#-------------------------------------------------------------------------
function Do-Initialize()
{
  $snapin = Get-PSSnapin | Where-Object { $_.Name -eq "iControlSnapIn" }
  if ( $null -eq $snapin )
  {
    Add-PSSnapIn iControlSnapIn
  }
  $success = Initialize-F5.iControl -HostName $g_bigip -Username $g_uid -Password $g_pwd;
  return $success;
}

#-------------------------------------------------------------------------
# Main Application Logic
#-------------------------------------------------------------------------
if ( ($g_bigip -eq $null) -or ($g_uid -eq $null) -or ($g_pwd -eq $null) )
{
  Show-Usage;
}

if ( Do-Initialize )
{
  Print-LocalTrafficSummary
}
else
{
  Write-Error "ERROR: iControl subsystem not initialized"
}
```

PowerShell has a handy CmdLet that helps with the displaying of tables.  Objects can be passed to the Format-Table Cmdlet and the table will be displayed on the console spaced out correctly with column headers and values.  The following code will pass an array of values returned from various internal function calls to Format-Table.  The magic here is that each of the objects should have the same properties so that the table will be displayed properly.  PowerShell has another nifty CmdLet that allows you to create custom objects and then define custom properties in that object.  The Add-Member CmdLet will do just this and later in this article, you will see the New-StatusObject function that returns an object to simulate the columns in each row of the generated table.

```
function Print-LocalTrafficSummary()
{
  ( (Get-VSStatus),
    (Get-PoolStatus),
    (Get-PoolMemberStatus),
    (Get-NodeStatus),
    (Get-iRuleStatus) ) | Format-Table
}
```

**Virtual Server Status**

The following function is called by the Print-LocalTrafficSummary function to return the object containing the Virtual Server Summary. In this function a new status object is created by the New-StatusObject function. A list of Virtual Server statuses is then returned from the iControl LocalLB.VirtualServer.get_object_status() method. The cached virtual server list, retrieved from the Get-VSList function, is passed in as a parameter to this method and a list of statuses for the virtual servers are returned. This list is iterated upon and passed into the Update-ObjectStatus function to update the counters on the status object for the supplied object status availability and enabled statuses. The filled object is then returned to the calling function.

```
function Get-VSStatus()
{
  $obj = New-StatusObject -name "Virtual Servers";

  $vs_statuses =
    (Get-F5.iControl).LocalLBVirtualServer.get_object_status( (Get-VSList) );
  foreach ($status in $vs_statuses)
  {
    Update-ObjectStatus $obj $status;
  }
  return $obj;
}
```

**Pool Status**

the Pool Status values are retreived with the following Get-PoolStatus function. It generates a status object named "Pools" and then requests a list of pool statuses from all the pools specified as default pools from within the virtual server list. The Get-VSPoolList does some trickery to query the list of default pools from all the Virtual Servers and truncate the list for the virtual servers that do not have a default pool associated with them. This list of pools associated to virtual servers is then iterated upon and their status is added to the status object which is then returned to the calling program.

```
function Get-PoolStatus()
{
  $obj = New-StatusObject -name "Pools";

  $pool_statuses =
    (Get-F5.iControl).LocalLBPool.get_object_status( (Get-VSPoolList) );
  foreach ($status in $pool_statuses)
  {
    Update-ObjectStatus $obj $status;
  }
  return $obj;
}
```

**Pool Member Status**

The Pool member status values are retrieved with the Get-PoolMemberStatus function. This function creates a new status object named "Pool members" and then makes a call to the LocalLB.PoolMember.get_object_status() iControl method to query the status of all pool members for the list of Pools that are associated with Virtual Servers that is retreieved from the Get-VSPoolList method. This list of object statuses is interated upon and the statuses are added to the status object with the Update-ObjectStatus function and then the status object is returned to the calling function.

```
function Get-PoolMemberStatus()
{
  $obj = New-StatusObject -name "Pool Members";
  $MemberStatAofA =
    (Get-F5.iControl).LocalLBPoolMember.get_object_status( (Get-VSPoolList) );
```

```
  foreach ($MemberStatA in $MemberStatAofA)
  {
    foreach ($MemberStat in $MemberStatA)
    {
      Update-ObjectStatus $obj $MemberStat.object_status;
    }
  }

  return $obj;
}
```

**Node Status**

The List of Nodes in the Local Traffic Summary table are the nodes that are referenced in at least one of the pools that are default pools for a virtual server. This function will generate a new status object named "Nodes" and then get a list of pool members from the globally cached virtual server pools returned from the Get-VSPoolList function. Since Pool members are full node servers (meaning address:port pairs), and we are only concerned with unique node addresses, a little busy work is needed to loop over all of the member definitions and make an array that contains only the unique IP addresses for the pool members. This will constitute the list of Nodes we will be reporting on. This list is stored in the local "node_list" variable. This list is then passed into the LocalLB.NodeAddress.get_object_status() method and we go through the same procedure of iterating through that list and updating the status objects values. Then the status object is returned to the calling function.

```
function Get-NodeStatus()
{
  $obj = New-StatusObject -name "Nodes";

  $IPPortDefAofA = (Get-F5.iControl).LocalLBPool.get_member( (Get-VSPoolList) );
  $node_list = $null;

  foreach($IPPortDefA in $IPPortDefAofA)
  {
    foreach($IPPortDef in $IPPortDefA)
    {
      if ( !(Is-InArray $node_list $IPPortDef.address) )
      {
        if ( $null -eq $node_list ) { $node_list = (, $IPPortDef.address) }
        else { $node_list += $IPPortDef.address; }
      }
    }
  }

  if ( $node_list.Length -gt 0 )
  {
    $NodeStatusA = (Get-F5.iControl).LocalLBNodeAddress.get_object_status($node_list);
    foreach ($status in $NodeStatusA)
    {
      Update-ObjectStatus $obj $status;
    }
  }

  return $obj;
}
```

**iRule Status**

For the iRule Status values, we are only interested in the iRules that are associated with a Virtual Server. This can be done by calling the Get-VSRuleList function which returns the cached list of iRules that are assocated with the cached list of Virtual Servers. For this type of object, there is no status outside of the total number of iRules, so we will iterate through the list and increment the totals by the number of iRules per virtual. All the other statuses are set to dashes as they are in the GUI.

```
function Get-iRuleStatus()
{
  $obj = New-StatusObject -name "iRules";

  $vs_list = Get-VSList;
  $VSRuleAofA = Get-VSRuleList;

  foreach ($VSRuleA in $VSRuleAofA)
  {
    $obj.Total += $VSRuleA.Length;
  }

  $obj.Available = "-";
  $obj.Unavailable = "-";
  $obj.Offline = "-";
  $obj.Unknown = "-";

  return $obj;
}
```

**Utility functions**

As I mentioned above, PowerShell has the Add-Member CmdLet that will allow dynamic creation of properties within an object. Essentially this function will create a structure with an "ObjectType", "Total", "Available", "Unavailable", "Offline", and "Unknown" members. The default value for the ObjectType is the passed in name value and all other statistics are set to 0.

```
function New-StatusObject()
{
  param(
    $name = $null
  );
  $obj = New-Object -TypeName System.Object;
  $obj | Add-Member -Type noteProperty -Name "ObjectType" $name;
  $obj | Add-Member -Type noteProperty -Name "Total" 0;
  $obj | Add-Member -Type noteProperty -Name "Available" 0;
  $obj | Add-Member -Type noteProperty -Name "Unavailable" 0;
  $obj | Add-Member -Type noteProperty -Name "Offline" 0;
  $obj | Add-Member -Type noteProperty -Name "Unknown" 0;

  return $obj;
}
```

The following function is used to update the status values within an object created with the New-StatusObject function. It is expecting a parameter of type iControl.Common.ObjectStatus to determine the status from. Since this structure has two values of availability_status and enabled_status, I've concatenated the values together in the switch statement and built cases for each combination. Depending on what this combination is, the relevant member of the status object is incremented.

```
function Update-ObjectStatus()
{
  param (
    $obj,
    $object_status
  );
  $obj.Total++;

  $avail = $object_status.availability_status;
  $enabled = $object_status.enabled_status;
  switch("$avail,$enabled")
  {
        # Available in some capacity
        "AVAILABILITY_STATUS_GREEN,ENABLED_STATUS_ENABLED" { $obj.Available++; }
        "AVAILABILITY_STATUS_GREEN,ENABLED_STATUS_DISABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_GREEN,ENABLED_STATUS_DISABLED_BY_PARENT" { $obj.Unavailable++; }

        # Object Not Available at the current moment
        "AVAILABILITY_STATUS_YELLOW,ENABLED_STATUS_ENABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_YELLOW,ENABLED_STATUS_DISABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_YELLOW,ENABLED_STATUS_DISABLED_BY_PARENT" { $obj.Unavailable++; }

        # Object is not available
        "AVAILABILITY_STATUS_RED,ENABLED_STATUS_ENABLED" { $obj.Offline++; }
        "AVAILABILITY_STATUS_RED,ENABLED_STATUS_DISABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_RED,ENABLED_STATUS_DISABLED_BY_PARENT" { $obj.Unavailable++; }

        # Object's availability status is unknown
        "AVAILABILITY_STATUS_BLUE,ENABLED_STATUS_ENABLED" { $obj.Unknown++; }
        "AVAILABILITY_STATUS_BLUE,ENABLED_STATUS_DISABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_BLUE,ENABLED_STATUS_DISABLED_BY_PARENT" { $obj.Unavailable++; }

        # Object is unlicensed
        "AVAILABILITY_STATUS_GRAY,ENABLED_STATUS_ENABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_GRAY,ENABLED_STATUS_DISABLED" { $obj.Unavailable++; }
        "AVAILABILITY_STATUS_GRAY,ENABLED_STATUS_DISABLED_BY_PARENT" { $obj.Unavailable++; }

        default { Write-Host "$_";}
  }
}
```

These functions are helpers around array management.  The Sanitize-Array will take as input a string array and return a string array minus any empty string elements.  The Is-InArray function will look through an array for a value and return whether or not the value exists in the specified array.

```
function Sanitize-Array()
{
  param (
    [string[]]$in
  );

  $out = $null;
  foreach ($item in $in)
  {
    if ( $item.Length -gt 0 )
    {
      if ( $null -eq $out ) { $out = ( , $item) }
```

```
      else { $out += $item }
    }
  }
  return $out;
}

function Is-InArray()
{
  param(
    [string[]]$in,
    [string]$val
  );
  $bFound = $false;

  foreach($v in $in)
  {
    if ( $v -eq $val )
    {
      $bFound = $true;
    }
  }

  return $bFound;
}
```

And the last set of helper methods are wrappers around the global cached objects for the virtual server list, virtual server pool list, and virtual server iRule list. If the values are null, they are queried. If non-null, the cached values are returned. The benefit of doing this allows for a reduction in iControl requests to be made for duplication information.

```
function Get-VSList()
{
  if ( $null -eq $g_vs_list )
  {
    $g_vs_list = (Get-F5.iControl).LocalLBVirtualServer.get_list();
  }
  return $g_vs_list;
}

function Get-VSPoolList()
{
  if ( $null -eq $g_vs_pool_list )
  {
    $g_vs_pool_list =
      Sanitize-Array (Get-F5.iControl).LocalLBVirtualServer.get_default_pool_name( (Get-VSList) );
  }
  return $g_vs_pool_list;
}

function Get-VSRuleList()
{
  if ( $null -eq $g_vs_rule_list )
  {
    $g_vs_rule_list =
      (Get-F5.iControl).LocalLBVirtualServer.get_rule( (Get-VSList) );
  }
  return $g_vs_rule_list;
}
```

**Running the Script**

Running this script with query all the Virtual Server, Pool, Pool Member, Nodes, and iRules states and report them in a table that is similar to the one in the Management GUI illustrated above.

```
PS C:\> .\LocalTrafficSummary.ps1 bigip username password

ObjectType                    Total          Available        Unavailable             Offline
----------                    -----          ---------        -----------             -------
Virtual Servers                   4                  2                  0                   0
Pools                             3                  2                  0                   0
Pool Members                      3                  2                  0                   0
Nodes                             1                  0                  0                   0
iRules                            1                  -                  -                   -
```

With a handful of iControl calls, key functionality can be repurposed for your own needs.  Be on the lookout for a future article discussing building a console based network map as available in the management GUI.

The full application can be found in the iControl CodeShare under PsLocalTrafficSummary.

Get the Flash Player to see this player.
20080626-iControlApps_2_LocalTrafficSummary.mp3

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com