Concurrent iControl Programming Explained

Joe Pruitt, 2008-20-11

iControl is a set of web services methods that can be called in any order by any number of clients. Typically an application is written with the plan that it is the only one accessing your devices. In some cases though, you may require the ability to have programs making configuration changes concurrently. In this article, I'll discuss the iControl lock methods that allow for each application to play well with each other.

All Methods Are Treated Alike

All iControl methods are separate entities. The iControl Portal is implemented as a single FIFO queue thus ensuring that two requests don't come in at the same time that would overlap each other. Often times, the internal methods implementing the interfaces are a sequence of methods on internal resources such as our configuration database. These sequence of calls are meant to be run in a particular order and overlapping iControl methods could mean that internal calls could be made out of order thus causing unpredictable things to happen.

Since the methods are thrown into a queue, this can cause some issues where calling applications are reliant on a sequence of iControl calls to be made to achieve a certain goal. This isn't a problem when you have one application making calls to your device, but in today's world of cloud computing and dynamic grid based networks, this is becoming less of the norm. Imagine the sequence of creating a virtual server with a pool and members. The sequence would be 1) create pool, 2) add pool members, 3) create virtual server, and 4) set default pool to newly created pool. Imagine what would happen if during that process another process came in and deleted the newly created pool. One or all of the subsequent methods in the first flow would fail.

This could be solved by building a transactional system into iControl but that currently available. So, what's a developer to do?

In the System.SystemInfo interface, there are several "lock" methods that you can use to "acquire" and "release" "locks". What is a "lock"? Well, a lock is a non-enforceable way to allow Program A" to tell everyone else that he is doing something and to wait until he's done. That way, "Program B" can inquire as to if it's OK for her to start doing something else and allow her to "wait" until "Program A" is finished doing whatever he's doing. There is nothing in the iControl server that blocks "Program B" from doing bad things to "Program A" but it does allow the developer a way to become multi-client aware.

The Methods

The methods are located in the System.SystemInfo interface. We have the **get_lock_list** method that returns a string list of all the acquired locks. The **get_lock_status** method takes an array of lock names and returns the time left on the lock as well as a user friendly comment used in the creation of the lock. The **acquire_lock** method can be used to create, or "acquire", a lock, and if you don't want to wait for the timeout to automatically release the lock, this can be done manually with the **release_lock** method.

```
String []
System.SystemInfo.get_lock_list(
  void
);

boolean
System.SystemInfo.acquire_lock(
    in String lock_name,
    in long duration_sec,
    in String comment
);

void
```

```
System.SystemInfo.release_lock(
    in String lock_name
);

struct System.LockStatus {
  string lock_name;
  long time_left;
  string comment;
};

LockStatus []
System.SystemInfo.get_lock_status(
    in String [] lock_names
);
```

Usage

The following code samples will build a PowerShell command line application allowing full control over the acquisition, release, and status of system locks. This program takes as input the bigip, username, and password as well as a subcommand and optional parameters of the lockname, timeout, and desc which are used depending on which sub command is isssued. The Write-Usage function displays the usage options for the application.

```
param (
  $g_bigip = $null,
  $g_uid = $null,
  $g_pwd = $null,
  $g_cmd = $null,
  $g_lockname = $null,
  $g_timeout = $null,
  $g_desc = $null
);

Set-PSDebug -strict;

function Write-Usage()
{
  Write-Host "Usage: ManageLocks.ps1 host uid pwd [options]";
  Write-Host "      options";
  Write-Host "      -------";
  Write-Host "      list                          - Get list of locks";
  Write-Host "      get     lockname              - Get status of a lock";
  Write-Host "      acquire lockname timeout desc - Create a new lock";
  Write-Host "      release lockname              - Release an existing lock";
  Write-Host "      wait    lockname              - Wait until lock is released";
  exit;
}
```

Initialization

As is with all of my iControl PowerShell scripts, validation is made as to whether the iControlSnapin is loaded into the current powershell context. The Initialize-F5.iControl cmdlet is then called to setup the connection to the BIG-IP for subsequent calls.

The main application logic checks for the passed in command and then passes control to either the Get-LockList, Get-LockStatus, Acquire-Lock, Release-Lock, or Wait-ForLock local functions which I will talk about below.

```
function Do-Initialize()
```

```
function Do-Initialize()
{
  if ( (Get-PSSnapin | Where-Object { $_.Name -eq "iControlSnapIn"}) -eq $null )
  {
    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 $null) -or ($g_cmd -eq $null) )
{
  Write-Usage;
}

if ( Do-Initialize )
{
  switch ($g_cmd.ToLower())
  {
    "list" {
      Get-LockList;
    }
    "get" {
      Get-LockStatus $g_lockname;
    }
    "acquire" {
      Acquire-Lock $g_lockname $g_timeout $g_desc;
    }
    "release" {
      Release-Lock $g_lockname;
    }
    "wait" {
      Wait-ForLock $g_lockname;
    }
    default {
      Write-Usage;
    }
  }
}
else
{
  Write-Error "ERROR: iControl subsystem not initialized"
}
```

Retrieving A List Of Locks

The Get-LockList function is simply a call to the iControl get_lock_list method. This returns a string list of currently active locks. This list is then passed into the Get-LockStatus function to query the status of each of the locks and display it to the console.

```
function Get-LockList()
{
  $Locks = (Get-F5.iControl).SystemSystemInfo.get_lock_list();
  Get-LockStatus $Locks;
```

```
}
```

Determining The Status Of A Lock

The Get-LockStatus function makes a call to the get_lock_status iControl method. This returns a LockStatus array, one entry for each requested lockname. This function loops over that returned list, extracts the relevant details to the lock and then writes it to the console.

You may notice the exception trap in the top of this function. This is there because the get_lock_status function throws a SOAPFault if you pass in a lock name that is not configured on the system. Instead of letting PowerShell puke out on the exception, I threw a trap in there that will call a local helper function that parses the fault to determine if it's an iControl fault. If it is an iControl fault, it will parse the fault data and populate an object with the data from the response. This is then presented to the console and control is returned.

```
function Get-LockStatus()
{
  param($lockname);

  trap {
    $ex = Parse-Exception $_.Exception;
    if ( $ex.PrimaryErrorCode -eq 16908289 )
    {
      Write-Host "Lock '$lockname' does not exist!";
    }
    else
    {
      Write-Host "Exception Occurred";
      Write-Host "Primary Error : " + ex.PrimaryErrorCode;
      Write-Host "Error String  : " + ex.ErrorString;
    }
    continue;
  }

  if ( $null -eq $lockname )
  {
    Write-Usage;
  }
  else
  {
    $LockStatusA = (Get-F5.icontrol).SystemSystemInfo.get_lock_status( $lockname );
    foreach ($LockStatus in $LockStatusA)
    {
      $name = $LockStatus.lock_name;
      $time_left = $LockStatus.time_left;
      $comment = $LockStatus.comment;
      Write-Host "Lock Name  : $name";
      Write-Host "Time Left  : $time_left";
      Write-Host "Description: $comment";
      Write-Host "----------------------";
    }
  }
}
```

Acquiring A Lock

This is a fairly simple function. The Acquire-Lock function takes as input a name for the lock, the duration you would like it for, and a user friendly description. It then calls the iControl acquire_lock method and then queries the status of the lock with the local Get-LockStatus function.

It is worth a mention here that if the lock already exists, this will be a no-op procedure meaning that the existing timeout will not be updated. To increase the timeout, you must release the lock and then acquire it again.

```
function Acquire-Lock()
{
  param($lockname, $timeout, $desc);
  if ( $null -eq $lockname )
  {
    Write-Usage;
  }
  else
  {
    if ( $null -eq $timeout ) { $timeout = 60; }
    if ( $null -eq $desc    ) { $desc = "60 Second Lock Acquired by ManageLocks PowerShell Script" };

    (Get-F5.iControl).SystemSystemInfo.acquire_lock($lockname, $timeout, $desc);
    Get-LockStatus $lockname;
  }
}
```

Releasing A Lock

The Release-Lock function takes as input the name of the lock and then calls the iControl release_lock method. This method has no return value and will not throw an exception. If the lock exists, then it will be released and if it does not exist, it is a no-op function.

```
function Release-Lock()
{
  param($lockname);
  if ( $null -eq $lockname )
  {
    Write-Usage;
  }
  else
  {
    (Get-F5.iControl).SystemSystemInfo.release_lock($lockname);
  }
}
```

Waiting For A Lock

The first thing a requestor of a lock would do if they find one is not available would be to wait until it is available and then acquire it. The Wait-ForLock function does just that, minus the acquiring part. It will query the status of the requested lock and as long as that lock exists and has a timeout greater than 0, it will sleep 5 seconds and retry. The function returns when the lock has been released and can be acquired.

```
function Wait-ForLock()
{
  param($lockname);

  trap {
    $ex = Parse-Exception $_.Exception;
    if ( $ex.PrimaryErrorCode -eq 16908289 )
    {
      Write-Host "Lock is available";
    }
    else
```

```
    else
    {
      Write-Host "Exception Occurred";
      Write-Host "Primary Error : " + ex.PrimaryErrorCode;
      Write-Host "Error String  : " + ex.ErrorString;
    }
    continue;
  }

  if ( $null -eq $lockname )
  {
    Write-Usage;
  }
  else
  {
    while($true)
    {
      $LockStatusA = (Get-F5.icontrol).SystemSystemInfo.get_lock_status( $lockname );
      $LockStatus = $LockStatusA[0];
      $name = $LockStatus.lock_name;
      $time_left = $LockStatus.time_left;
      $comment = $LockStatus.comment;
      if ( $time_left -gt 0 )
      {
        Write-Host "$time_left seconds remaining on lock $lockname...";
        Start-Sleep 5
      }
      else
      {
        Write-Host "Lock is available";
        break;
      }
    }
  }
}
```

Helper Functions

For fun, I threw in a little helper function here, mainly because I haven't done so yet in any of my PowerShell applications. It will parse the exception's internal message text and if it finds it is an iControl exception, it will parse the lines and create an object with PrimaryErrorCode, PrimaryErrrorCodeOct, SecondaryErrorCode, and ErrorString members for later interrogation.

```
function Parse-Exception()
{
  param($ex);
  $obj = New-Object -TypeName System.Object;

  if ( $ex.Message.Contains("urn:iControl") )
  {
    $obj | Add-Member -Type noteProperty -Name "IsiControlError" $true;
  }
  else
  {
    $obj | Add-Member -Type noteProperty -Name "IsiControlError" $false;
  }

  $sr = New-Object System.IO.StringReader($ex.Message);
```

```
  $line = $sr.ReadLine();
  while($null -ne $line)
  {
    if ( $line.Contains("primary_error_code") )
    {
      $parts = $line.Split( (, ':'));
      $v = $parts[1].Trim();
      $vals = $v.Split( (, ' '));
      $p_dec = $vals[0];
      $p_oct = $vals[1].Trim('()');

      $obj | Add-Member -Type noteProperty -Name "PrimaryErrorCode" $p_dec;
      $obj | Add-Member -Type noteProperty -Name "PrimaryErrorCodeOctal" $p_oct;
    }
    elseif ( $line.Contains("secondary_error_code") )
    {
      $parts = $line.Split( (, ':'));
      $v = $parts[1].Trim();
      $obj | Add-Member -Type noteProperty -Name "SecondaryErrorCode" $v;
    }
    elseif ( $line.Contains("error_string") )
    {
      $parts = $line.Split( (, ':'));
      $v = $parts[1].Trim();
      $obj | Add-Member -Type noteProperty -Name "ErrorString" $v;
    }
    $line = $sr.ReadLine();
  }
  return $obj;
}
```

Conclusion

With these four methods, you can be well on your way to building parallelism into your iControl application. It's not perfect and bulletproof, but it should get you to where you need to go.

For the full PowerShell application, check out the PsIcontrolConcurrency addition to the iControl CodeShare.

Get the Flash Player to see this player.
20081120-iControlApps-16-ConcurrentiControl.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