EMC CLARiiON® CLI Administration and Ruby: A Powerful Combination

EMC Proven™ Professional Knowledge Sharing

Wagner H. Ikeda
wagner.ikeda@eds.com
wagner.ikeda@gmail.com
Infrastructure Specialist
Electronic Data Systems

Table of Contents

Introduction	3
The strengths of NAVICLI	3
Why Ruby?	4
Sample code download	6
Script environment and requisites	6
Design	7
Implementation: data modeling	7
Implementation: data collection and parsing	9
Implementation: excel report generation	11
Implementation: Query tool using IRB	14
Conclusion	20
Appendix A – References	20
Author's Biography	20

Introduction

The EMC Navisphere® Command Line Interface (NAVICLI) implements a complete set of commands to query and modify the configuration of a CLARiiON system. When we combine NAVICLI with a high level scripting language like Ruby, we can achieve very interesting results.

This article will explore how we can use NAVICLI and Ruby to resolve two problems:

1) automatic generation of an Excel spreadsheet documenting the current CLARiiON configuration
2) implementation of an interactive, text-based, CLARiiON configuration query utility

The first tool generates and maintains configuration documents much more effectively. No need for cutting, pasting, importing/exporting of comma-separated values (CSV) files, or even long typing sessions.

When a storage system has hundreds of storage objects (LUNs/MetaLUNs, Storage Groups, Raid Groups), the process of navigating through the object hierarchy in a graphical interface is not effective. With the second tool, some configuration details of a CLARiiON can be verified quickly and interactively.

The strengths of NAVICLI

The Navisphere Command Line Interface (NAVICLI) is very powerful, implementing a complete set of commands to perform CLARiiON administration. It is typically used inside scripts that perform automated CLARiiON system administration tasks. It is also used to perform some administration tasks which cannot be executed from the web-based administration interface.

However, it is not suitable for interactive use, unless used in combination with some UNIX utilities such as alias, grep, awk and sed, among others. NAVICLI commands sometimes require a long list of parameters. The output has many lines that make it difficult to get quick results from the NAVICLI command alone. We need to combine NAVICILI with a scripting language to get the best it has to offer.

Why Ruby?

I tried three different approaches to automate UNIX system administration tasks. The first was standard shell scripting with Bourne Shell (sh), C Shell (csh) and Korn Shell (ksh). I wasn't satisfied. There are some inconsistencies which make the learning, usage and maintenance of shell scripts difficult. Let's look at the simple ksh script below:

```
#!/bin/ksh
v=""
if [ "$v" = "" ]; then
        ls -al
fi
for var in `ls`; do
        echo $var
done
```

Why do the *if* block ends with a *fi*? Why do the *for* block end with *done* and not with *rof*? Why can't the variable *var* in the *for* statement have a prepended *$,* and why must the same variable *var* inside the *for* block have a prepended *$*?

The creators of the ksh had a good reason to define the syntax that way. But it does not align with my thinking process. Sometimes, I have to take a look at a shell script and perform some maintenance; but if I need to code a script from scratch, shell scripting is not my choice.

Next, I tried Perl. I found it very elegant and became my scripting language of choice. The ksh example above could be translated into Perl this way:

```
#!/usr/bin/perl
$v = "";
system("ls -al") if $var ne "";
foreach $var (`ls`) {
  print $var;
}
```

Perl has a good learning curve, but maintenance sometimes can suffer from cryptic code. Perl is very powerful, its syntax makes it possible to write the same algorithms in many different ways. Unfortunately, this can lead to code which is difficult to understand. This is a common problem with programming languages.

I learned Perl at version 4.036 that did not implement object oriented programming (OOP). I never used Perl as an OOP language because I also was doing fewer system administration tasks and did not have to write complex scripts at all.

After many years, I became involved with networked storage administration. Many of the tasks I o perform demand automation. I continued to use Perl, but felt that I had to evaluate other possibilities. So I decided to investigate Ruby. Everyone was talking about this language thanks to the rapid web application development platform Ruby on Rails. My first impressions were very positive and I was surprised by:

- Ruby's concise syntax
- how quickly the language allowed me to transform thoughts into working code
- the Interactive Ruby Shell (IRB) that makes it possible to test code and use existing code to implement an interactive query system.

Here's our simple example coded in Ruby:

```
#!/usr/bin/ruby
v = “”
Kernel.system("ls -al") if v != ""
`ls`.each { |f| puts f }
```

Ruby is an OOP language; every piece of data in Ruby is an object. For a reference to OOP, please visit the links in the Appendix A.

The construct *Kernel.system(“ls -al”)* invokes the method *system* from the object *Kernel*. It executes the command passed as argument, in the same way as the ksh and the Perl script detailed above.

There is also the very interesting code *`ls`.each { |f| puts f }* , let's examine it.

Backticks contain the command *ls*. The backticks instruct the Ruby interpreter to execute the command *ls*. The command returns a String object which implements the method *each*, this OOP construct is known as the iterator. The method *each* requires a block of code, which in this case will process every line in the String, with each line being presented as a variable, *f*.

We can try this code, or just one portion of it with the IRB:

```
$ irb
>> `ls`
=> "1.ksh\n1.pl\n1.rb\n"
>> `ls`.class
=> String
>> `ls`.each { |f| puts f }
1.ksh
1.pl
1.rb
=> "1.ksh\n1.pl\n1.rb\n"
```

Sample code download

To facilitate the understanding of this article, a basic functioning version of the scripts I developed and sample CLARiiON configuration data is available for download at http://www.k33p34.net/EMC The reader is encouraged to try it and provide feedback.

Script environment and requisites

Environment used to design, code, test and run the scripts:

- NAVICLI is installed along with the required Java environment in the management platform.
- I used a laptop running Windows XP Pro SP 2 at work, a MacBookPro and an iMac G5 both running MacOSX 10.5.1 at home.
- To make things easier, I installed Cygwin at my work laptop. Cygwin is a Linux-like environment for Windows that provides the standard Unix command line tools, and also the Ruby implementation.
- An implementation of Ruby for Windows is also available and can be used to run these scripts.
- My laptop at work is used to generate Excel that requires Microsoft Office. The report generation script relies on the Ruby library Win32OLE which comes by default with Ruby for Windows platforms.

The mandatory components are: NAVICLI to collect CLARiiON configuration data; Ruby on any platform to process the configuration data; and a Windows system with Microsoft Office, to generate Excel reports.

Design

Generating an Excel file to document a CLARiiON configuration, and providing an interactive CLARiiON configuration query interface involve four different processes:

1) data modeling
2) data collection and parsing
3) Excel report generation
4) Query tool implementation using IRB

The basic design starts with data modeling, where the classes are defined. A class is an OOP construct where data and associated methods to process the data are grouped together. An instance of a class is an object.

The data collection process uses standard NAVICLI to obtain the CLARiiON configuration. The configuration data is stored in text files containing unprocessed NAVICLI output. The text files are parsed to generate persistent data structures using YAML (Yet Another Markup Language). The stored configuration data can later be read and used by Ruby, Perl, Python, Java or Javascript programs.

With the serialized data, a script opens and displays an instance of Excel. After processing the configuration data, the script starts to fill in cells in a spreadsheet, which shows a matrix containing Raid Group, Storage Group, LUN, MetaLUN information.

Implementation: **data modeling**

The classes defined to represent CLARiiON configuration items are:

- LUN: stores LUN configuration data
- MetaLUN: stores MetaLUN configuration data
- RaidGroup: stores RaidGroup configuration data
- StorageGroup: stores StorageGroup configuration data
- CLARiiON: stores CLARiiON configuration data and defines methods to query a list of objects or a specific object

Implementation of a class LUN:

```
class LUN
  @id
  @name
  @rg
  @rl
  @c_owner
  @d_owner
  @private
  @capacity

  attr_reader :id, :name, :rg, :rl, :c_owner, :d_owner, :private, :capacity
  def initialize(id, name, rg, rl, c_owner, d_owner, private, capacity)
    @id = id
    @name = name
    @rg = rg
    @rl = rl
    @c_owner = c_owner
    @d_owner = d_owner
    @private = private
    @capacity = capacity
  end
end
```

In the beginning of the class, we define the variables that store LUN id, LUN name, Raid Group that contains the LUN, raid level, current and default SP owner, private flag (if the LUN is private or not) and LUN capacity.

The class defines an *initialize* method that is called every time a new LUN object is created. The attr_reader function defines that all of the listed parameters can be accessed using methods. So for example, to get the capacity of a LUN:

```
puts LUN1.capacity
#this statement prints the capacity of the LUN object defined by LUN1
```

MetaLUN, RaidGroup and StorageGroup classes are defined in a similar way.
The CLARiiON class is described below in Implementation: Query Tool using IRB.

Implementation: data collection and parsing

The data collection process runs at the management station, using the standard NAVICLI and Java NAVICLI (to gather MetaLUN information).

The commands below are used to collect LUN, Raid Group, Storage Group and MetaLUN data:

```
$ navicli –h <CLARiiON ip address> getLUN > LUN.txt
$ navicli  –h <CLARiiON ip address> getrg > rg.txt
$ navicli  –h <CLARiiON ip address> storagegroup –list > sg.txt
$ java –jar navicli.jar –address <CLARiiON ip address> metaLUN –list > meta.txt
```

The files with the collected data are transferred to the computer where the data will be parsed and used. The parsing process involves the use of four different scripts:

- LUN.rb to parse LUN data
- meta.rb to parse MetaLUN data
- rg.rb to parse Raid Group data
- sg.rb to parse Storage Group data

We run the scripts to execute the parsing, we providing the appropriate data:

```
$ ruby LUN.rb < LUN.txt
$ ruby meta.rb < meta.txt
$ ruby rg.rb < rg.txt
$ ruby sg.rb < sg.txt
```

The meta.rb script follows on the next page.

The meta.rb script is listed below:

```
require 'myarray'
require 'metaLUN'
require ‘yaml’
LUNs = Array.new
meta = Array.new
for line in STDIN.readlines do
  if (line =~ /MetaLUN Number:\s+(\d+)/)
    m_number = $1
  end
  if (line =~ /MetaLUN Name:\s+(\w+)/)
    m_name = $1
  end
  if (line =~ /Current Owner:\s+(SP (A|B))/)
    m_c_owner = $1
  end
  if (line =~ /Default Owner:\s+(SP (A|B))/)
    m_d_owner = $1
  end
  if (line =~ /Total User Capacity.*\s\d+\/(\d+)/)
    m_cap = $1
  end
  if (line =~ /Number of LUNs:\s+(\d+)/)
    m_nLUNs = $1
  end
  if (line =~ /LUN Number:\s+(\d+)/)
    LUNs.push($1)
  end
  if (line =~ /^\s+$/)
    ml = MetaLUN.new(m_number, m_name, m_c_owner, m_d_owner, m_cap, m_nLUNs,
LUNs)
    meta.push(ml)
    LUNs = Array.new
  end
end
open('METALUN', 'w') { |f| YAML.dump(meta, f) }
```

This script uses two Arrays: LUNs to store the component LUNs of a MetaLUN, and meta which stores all MetaLUNs defined at a CLARiiON.

The script loops, looking for patterns at the NAVICLI output and populating the meta Array.
When the NAVICLI output ends, the meta Array is written to disk, using the YAML format.
The other three parsers work the same way.

Implementation: excel report generation

We can extract useful information and generate reports with the YAML files generated by the parsers. Using Ruby, we can populate automatically an Excel spreadsheet using the library Win32OLE.

The script clar_doc.rb which generates the Excel spreadsheet is detailed on the next page.

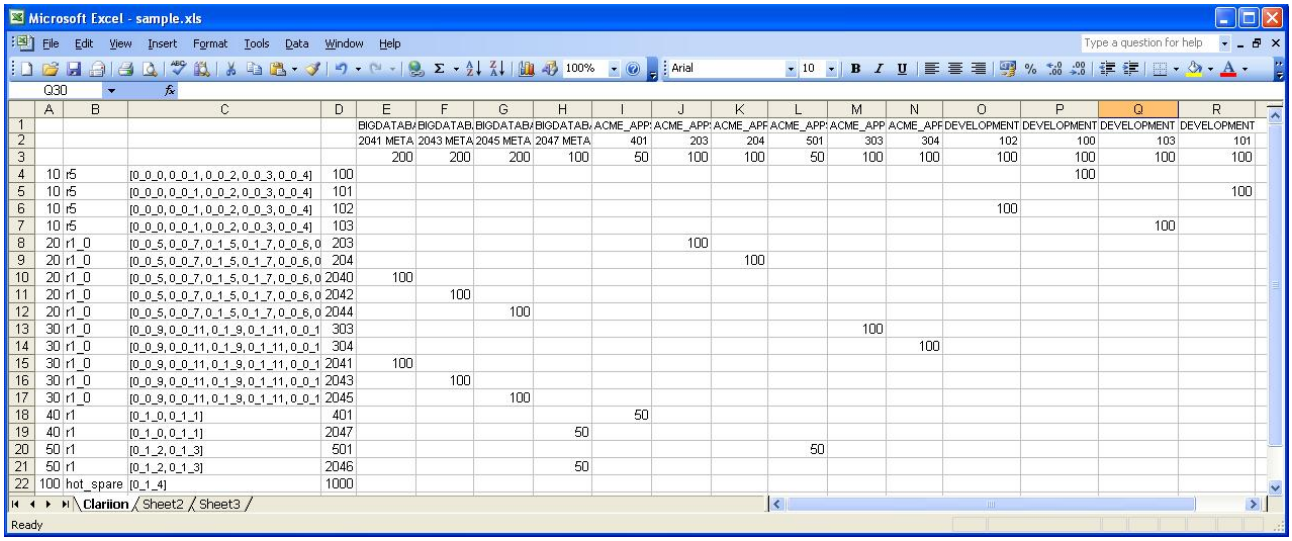
```
require 'LUNs'
require 'metaLUN'
require 'raidgroup'
require 'storagegroup'
require 'CLARiiON'
require 'win32ole'

x1 = WIN32OLE.new('Excel.Application')
x1.Visible = 1
wb = x1.Workbooks.Add
ws = wb.Worksheets(1)
ws.Name = 'CLARiiON'
row = 4
col = 1 # not 0 !!! starts at 1

LUN_row = Array.new
LUNs = Hash.new
CX700.rg.each { |rg|
  LUNs[rg.id] = rg.LUN
}
LUNs.keys.sort { |x, y| x.to_i <=> y.to_i }.each { |l|
  rt = CX700.raidgroup(l.to_i).type
  disks = CX700.raidgroup(l.to_i).disks.putsc
  LUNs[l].sort { |x, y| x.to_i <=> y.to_i }.each { |ll|
    ws.Cells(row, col).Value = l   #ws.Cells(row, col).Value = value # this is used to populate
    ws.Cells(row, col + 1).Value = rt    # the spreadsheet
    ws.Cells(row, col + 2).Value = disks
    ws.Cells(row, col + 3).Value = ll
    LUN_row[ll.to_i] = row
    row += 1
  }
}
col = 6
```

```
CX700.sg.each { |sg|
  sg.LUNs.each { |l|
    ws.Cells(1, col).Value = sg.name
    ws.Cells(2, col).Value = l
    if not m = CX700.metaLUN(l)
      cap = CX700.lu(l).capacity.to_i/1024
      ws.Cells(2, col).Value = l
      ws.Cells(3, col).Value = ws.Cells(LUN_row[l.to_i], col).Value = cap.to_s
    else
      first = true
      m.members.each { |l|
        ws.Cells(2, col).Value = l.to_s + " META"
        cap = CX700.lu(l).capacity.to_i/1024
        if first
          ws.Cells(3, col).Value = cap.to_s
          first = false
        else
          ws.Cells(LUN_row[l.to_i], col).Value = cap.to_s
        end
      }
    end
    col += 1
  }
```

Below, an example of documentation generated with the clar_doc.rb script.



Implementation: query tool using IRB

The implementation of the query tool depends on the CLARiiON class:

```
require 'LUNs'
require 'metaLUN'
require 'raidgroup'
require 'storagegroup'
require 'c_array'
require 'myarray'
require ‘yaml’
class CLARiiON
  @rg
  @LUN
  @meta
  @sg

  def initialize
    @rg = open('RG') { |f| YAML.load(f) }
    @LUN = open('LUN') { |f| YAML.load(f) }
    @meta = open('METALUN') { |f| YAML.load(f) }
    @sg = open('SG') { |f| YAML.load(f) }
  end

  attr_reader :rg, :LUN, :meta, :sg
```

```
def lu(LUN = "")
    if (LUN == "")
      self.LUN.collect { |l| l }
    else
      self.LUN.collect { |l| return l if (l.id == LUN.to_s) }
      return nil
    end
  end
  def metaLUN(meta = "")
    if (meta == "")
      self.meta.collect { |m| m }
    else
      self.meta.collect { |m| return m if (m.id == meta.to_s) }
      return nil
    end
  end
  def raidgroup(rg = "")
    if (rg == "")
      self.rg.collect { |r| r }
    else
      self.rg.collect { |r| return r if (r.id == rg.to_s) }
      return nil
    end
  end
  def storagegroup(sg = "")
    if (sg == "")
      self.sg.collect { |s| s }
    else
      self.sg.collect { |s| return s if (s.name == sg) }
      return nil
    end
  end
end

CX700 = CLARiiON.new
```

Please note that the initialization of a CLARiiON object involves reading the YAML files “RG”, “LUN”, “METALUN” and “SG” . These files contains data parsed from the output of NAVICLI and must be present to instantiate a CLARiiON object.

The class also implements methods to query the list of LUNs, MetaLUNs, Storage Groups and RaidGroups. If the query method has no arguments, it will return the complete list of corresponding objects. If an argument is supplied, it will return an object corresponding to the argument or nil. Finally the class instantiates a CLARiiON object named CX700.

Example of the use of CLARiiON class:

```
$ irb -r CLARiiON.rb
>> CX700.storagegroup
=> [#<StorageGroup:0x58aa70 @name="BIGDATABASE", @LUNs=["202", "201", "200", "500"]>, #<StorageGroup:0x58a69c @name="ACME_APPSERV", @LUNs=["401", "203", "204", "501", "303", "304"]>, #<StorageGroup:0x58a2dc @name="DEVELOPMENT", @LUNs=["102", "100", "103", "101"]>]
>> CX700.storagegroup("DEVELOPMENT")
=> #<StorageGroup:0x58a2dc @name="DEVELOPMENT", @LUNs=["102", "100", "103", "101"]>
>> CX700.storagegroup("SAP PRODUCTION")
=> nil
```

When we invoke the IRB using the option -r <file>, the IRB will load and evaluate the file. Since the parameter is CLARiiON.rb, it loaded and executed this file so the object CX700 is available for interactive use. Please observe that the required YAML files must be available.

In the first example, the storagegroup method was used with no parameters, so the list of all storage group objects contained in the CX700 object were returned.

In the second example, we want more information about the “DEVELOPMENT” storage group.

In the last example we query information about the “SAP PRODUCTION” storage group, which does not exist, so the query returns nil.

In Ruby, it is possible to modify the implementation of every class. It can result in some unexpected behaviour:

```
$ cat weird.rb
class Fixnum
  def +(arg)
    0
  end
end
$ irb
>> 1 + 1
=> 2
>> load "weird.rb"
0
=> true
>> 1 + 1
=> 0
```

We can modify the behaviour of the operator + making all of the sum operations to return 0!
But we will not make dangerous modifications, like this. We'll use this Ruby feature to our advantage.

In the storage group query example above, the returned objects contains a list of LUNs represented by an Array. An Array is a container data structure, which is represented in Ruby by the notation: ["element1", "element2", "element3"].

Ruby defines Array as a standard class. It would be very useful if we modify the behaviour of the Array class, making it aware of the attributes of a LUN, like raid level, number of spindles (disks), and so on.

These modifications of the Array class are implemented in the file c_array.rb:

```
class Array
  def cap
    list = Hash.new
    self.each { |l| list[l] = CX700.lu(l).capacity }
    return list
  end

  def spindles
    list = Hash.new
    self.each { |l| list[l] = CX700.raidgroup(CX700.lu(l).rg).disks.length }
    return list
  end

  def spindles_gt(n)
    list = Array.new
    self.each { |l|
      a = CX700.raidgroup(CX700.lu(l).rg).disks.length
      list.push(l) if (a > n)
    }
    return list
  end

  def members
    return CX700.metaLUN(self).members if self.length == 1
    list = Hash.new
    self.each { |l| list[l] = CX700.metaLUN(l).members }
    return list
  end
```

```
  def raidlevel(level)
    if level == 10
      level = "1_0"
    end
    level = "r" + level.to_s
    list = Array.new
    self.each { |l|
      if (CX700.raidgroup(CX700.lu(l).rg).type == level)
        list.push(l)
      end
    }
    return list
  end
end
```

With this modification to the Array class, we can make queries like:

```
>> CX700.storagegroup("ACME_APPSERV").LUNs.spindles
=> {"303"=>8, "501"=>2, "204"=>8, "304"=>8, "203"=>8, "401"=>2}
>> CX700.storagegroup("ACME_APPSERV").LUNs.raidlevel(10)
=> ["203", "204", "303", "304"]
>> CX700.storagegroup("ACME_APPSERV").LUNs.spindles_gt(5)
=> ["203", "204", "303", "304"]
```

In the first example, we want to know how many spindles for every LUN contained in the storage group “ACME_APPSERV”.

The next example lists all the LUNs in the storage group that are raid level 10, and in the last example we ask for a list of all LUNs in this storage group with more than 5 spindles.

Conclusion

The combination of NAVICLI and Ruby allowed us to code interesting tools that save a great deal of time and get the job done. The classes and scripts presented in this article can be modified easily, serving as a base for future improvements.

Ruby integrates well with database management systems (DBMS), so it is easy to evolve the presented scripts to use a DBMS like MySQL or Oracle, instead of the use of a simple file with YAML.

The concepts illustrated in this article should be deployed to help us face the challenges involved in administering large scale networked storage environments. As you will learn, Ruby can be used with great results with administrative command line interfaces used to manage other kind of systems, like the DMX family and SAN switches.

I hope this article helps you to decide to use Ruby as a tool to improve your work.

Appendix A – References

OOP: http://en.wikipedia.org/wiki/Object-oriented_programming
Ruby: http://www.ruby-lang.org/en/
Ruby for Windows: http://www.ruby-lang.org/en/downloads/
Cygwin: http://www.cygwin.com
YAML: www.yaml.org

Author's Biography

In 1993, Universidade Estadual de Campinas (www.unicamp.br) granted me a B.Sc. in Computer Engineering. I have 15 years of experience in IT, with extensive contact with UNIX system administration, TCP/IP networking and Internet services, consulting and training. In the past few years, I've been working with storage systems. I worked as a consultant for private companies, government agencies and educational institutions. In 2005, I delivered CLARiiON training for EMC Global Education in the U.S.A. and Brazil. I joined EDS in 2007. Currently I have an EMC Technology Foundations certification.