ns4
Configuration Management Tool

by Chris Mason (chris@noodles.org.uk)

User Guide
v4.3

24 November 2009

Contents

1. Introduction

ns4 is a powerful configuration management tool that allows running commands on just about anything that has a CLI. Commands are defined within a configuration file, and when they are executed, the output can be sent to a series of different types of server (transports) for archiving. As well as archiving configurations, it allows scripts to be run on nodes; this allows configurations to be applied en masse and allows conditional logic so different bits of scripts are run on different nodes.

ns4 started off as a tool to help in the automated backup of Cisco routers within a Service Provider's network. The initial requirement of Cisco routers soon exploded into a series of other pieces of equipment made by numerous vendors. Once simplistic configuration backup was sorted I started to look at how to automate tasks within our network and the scripting capabilities were added to allow me to deploy configuration changes en masse. This allowed people to write scripts which allowed them to audit their entire network by collecting information from the nodes within the network and format it into a tabular view for easy auditing.

The main design goal of ns4 was to be stable and extremely fast - when you have 5,000 nodes you wish to process you need to be very efficient. To be able to process that many devices efficiently I adopted a pre-forking model within the code which allowed me to have an array of the nodes that each pre-forked child was able to process a batch of. Once the children have finished processing a node it would send a status back to the parent through a Unix domain socket. I devised an algorithm which determines the loading across pre-forked children and how many children to pre-fork to provide optimal loading.

Be warned though, as ns4 is extremely powerful, it has the potential to do things very quickly so you need to be careful as you could trash a network quite quickly if you try running the wrong commands. I claim no responsibility for any network losses you experience as a result of running this tool.

You can always obtain the latest version of ns4 from http://www.noodles.org.uk/ns4.html or by going to http://www.freshmeat.net/projects/ns4.

2. Minimum Requirements

ns4 doesn't really have any minimum requirements - except a Unix environment. ns4 is written in Perl so it requires a Perl runtime environment to be able to run, but 99% of UNIX systems come with Perl as standard or can have Perl installed quite easily. On top of the default Perl installation, ns4 requires the following mandatory additional Perl modules to be installed:

Expect
YAML::Syck
File::Path

There are also additional optional modules which you will need to install depending on your configuration. Alerts and Transports use different modules, but as you won't be using every different type of Transport then you won't need every Perl module to be installed.

alert_smtp	Net::SMTP
alert_http	LWP::UserAgent
transport_ftp	Net::FTP
transport_sftp	Net::SSH2

Finally there are a few more Perl modules which are optional depending on the configuration:

Sys::Syslog
MIME::Base64
Net::SNMP

Without the Expect module then ns4 would be nothing, it uses this module extensively to be able to login to routers and switches and run commands. However, be aware that the Expect module can sometimes be a complete arse to install on some UNIX systems, but persistence usually works in the end (and the ability to hack Makefiles)!

Although I specifically mention UNIX, ns4 is not limited to UNIX alone and was mostly developed under Linux. There is also support under Microsoft Windows, not natively, but under Cygwin which allows you to have a UNIX environment within Windows which provides a complete run-time environment for ns4. Further information on Cygwin can be found at http://www.cygwin.com.

3. Installation

Within the ns4 distribution you will find the following files:

ns4	This is the main ns4 binary which contains all the complex stuff and black magic.
cfg/cartridges.yaml	This is a sample cartridge file which contains some of the cartridges out of this document.
cfg/ns4.yaml	This is a sample configuration file which contains the nodes which are used within the examples in this document.
doc/ns43-ug.pdf	This is the ns4 user guide which you are currently reading.
doc/changelog.txt	This is the detailed change log which I maintain from version to version. It contains all the major changes from one release to another.
scripts/meshping.nss	This is an example script which can be used on Cisco routers that use IS-IS as their IGP. It attempts to do a connectivity test from the node it is being run on to all the other nodes within the IS-IS domain.
scripts/cdiff.nss	This is an example script which shows an easy way to mimic basic RANCID functionality through an ns4 script. It performs a functional diff of a configuration files from routers. You setup a local repository and then it will email out details of all the configuration changes. It can easily be adapted to mimic full RANCID functionality.

Note: The files in **red** are mandatory files that are required for ns4 to work on a system. The other files are optional and are not absolutely necessary.

Although the ns4 binary can be situated anywhere on your system, the default path for "cartridges.yaml" and "ns4.yaml" is within "/etc/ns4". This default can be changed with the "-C" and "-R" command line options or through environment variables.

If you are deploying ns4 into a shared user environment you are going to need to ensure you restrict the whole world from running ns4. Normally you need a username and password before you are able to login to a node, but if ns4 is incorrectly protected then anyone who has a shell account on the UNIX box could potentially have full access to your data network.

Not only do you need to ensure there is adequate protection around the ns4 binary, but you also need to ensure you have protected the configuration files from prying eyes. The encoding for passwords within the configuration file is just that - encoding, it is not encryption and it isn't meant to be and is also optional. It is there to stop people looking over your shoulder and reading your passwords when you are editing the configuration file.

If you are planning to be the only user running ns4 and you have it self-contained within your home directory then you shouldn't have any concerns. However, you do need to ensure you have set sensible file permissions of 700 for the ns4 binary and 600 for the configuration files.

From experience the best way to deploy ns4 in a shared user environment is by creating an ns4 group:

```
$ groupadd ns4
```

You would then ensure that the ns4 binary and configuration files are only readable by either root or people within the ns4 group:

```
$ chown root:ns4 /usr/local/bin/ns4
$ chmod 750 /usr/local/bin/ns4
```

```
$ chown root:ns4 /etc/ns4/cartridges.yaml
$ chmod 640 /etc/ns4/cartridges.yaml

$ chown root:ns4 /etc/ns4/ns4.yaml
$ chmod 640 /etc/ns4/ns4.yaml
```

This will ensure that only people within the ns4 group have access to execute the ns4 binary or read the configuration files. You can then add users to the ns4 group when you wish to grant them access:

```
$ useradd -G ns4 <user>
```

Note: The above syntax may be different on different flavours of UNIX. The above syntax was verified against Linux.

If you have a scenario where different sets of users have access to different nodes then you can create multiple groups and multiple configuration files with different ownership.

4. Command Line Arguments

We are going to start off by ignoring configuration files and cartridges and start getting a feel of what ns4 can do. We are going to assume we have already written a configuration file and a cartridge for a small network consisting of Cisco routers.

If we run ns4 without any arguments we are going to get the following usage screen:

```
ns4 v4.3.6
Configuration Management Tool
url http://www.noodles.org.uk/ns4.html
by Chris Mason <chris@noodles.org.uk>

Usage: ns4 <criteria> <action> [options]
 Criteria:
  -a                  all nodes
  or -n 're'          node pattern
  or -N 'file'        node list file
  or/and -t 're'@     tag pattern
 Actions:
  -r '[type]'         node summary report
   where 'type' is:
    'byproxy'          group by proxy (default)
    'bytag'            group by tag
    'raw'              raw output
    'detail'           detailed output
  or -c 'x[,x,..],y'  command = 'x', output file = 'y'
  or -l 'x,y'         list file = 'x', output file = 'y'
  or -s 'x[,a=x,..]'  script file = 'x', variables ('a=x', etc)
  or -d '[file]'      execute commands from configuration
  or -x               test mode (verify node login only)
  or -p '[logfile]'   simple snmp poller
 Options:
  -C 'file'           alternative configuration file
  -R 'file'           alternative cartridges file
  -f '[file]'         create failed node list file
  -A                  enable alerting of failures
  -V                  output version information
  -v@                 verbosity level (max 4)
```

4.1. Criteria

You must specify a set of criteria so ns4 knows what nodes you want to process. There are four separate ways you can do this and each has its own advantages:

-a This option allows you to select all nodes which are defined within the configuration file. It is a synonym of "**-n** '.+'".

-n 're' This option allows you to specify a regular expression to select nodes. You can only specify this option once, but you have the power of Perl extended regular expressions to be able to be as flexible as you want.

- **-n** '.+' Include all nodes.
- **-n** '^PE' Include all nodes which begin with "PE".
- **-n** '^(PE|PP|RR)' Include all nodes which begin with "PE", "PP" or "RR".
- **-n** '01$' Include all nodes which end in "01".
- **-n** '^(?!.*PE)' Include all nodes which don't contain "PE". This uses a negative look ahead assertion.

-N 'file'

This option allows you to specify a list of node names within a file. They will be treated as literal values and won't get evaluated as regular expressions. Spaces or lines beginning with "#" are ignored and it will expect to see a single node name on each line. The following can be used as an example to select five nodes:

```
PE01
PE02
PP07
RR03
CE09
```

Any values which don't match a valid node name will be ignored.

-t 're'

This option allows you to specify a regular expression to select nodes which are associated with certain tags. Tags allow you to group certain types of nodes together and a node can have as many tags as you want. Tags are defined within the configuration file.

-t '.*'	Include all nodes (this is the default if you omit this option). We use a "*" here as you don't need to set a tag on a node.
-t '^PE$'	Include all nodes which are associated with the "PE" tag.
-t 'IP'	Include all nodes which are associated with any tag that has the word "IP" in it.
-t '^(?!.*IP)'	Include all nodes which are associated with any tag that doesn't have the word "IP" in it. This uses a negative look ahead assertion.
-t '^$'	Include all nodes which are not associated with a tag.

You may specify as many tag options as you like and they are all ANDed together so they must all be true for a node to be selected.

You are only allowed to specify either "-a", "-n" or "-N" options as they are mutually exclusive, but you can specify the "-t" option with any of them to limit down the scope of selected nodes. You don't have to specify either the "-a", "-n" or "-N" options as you can just select nodes using the "-t" option.

Note: All node names and regular expressions are treated as case-insensitive so there is no difference between "PE" and "pe".

4.2. Actions

Once you have selected the criteria you must select an action to perform on the nodes you have selected. Each action is mutually exclusive with every other action so you can only specify a single action.

You have the ability within the configuration file to specify a list of commands as well as having the ability to run adhoc commands from the command line. The reason why you can do both is for flexibility. Normally you would configure your configuration backup commands within the configuration file, i.e. "show running-config" for your Cisco routers. You would then schedule ns4 to run through a crontab using the "-d" option and your configurations are backed up, but you also have the ability of running adhoc commands or scripts on all or a selection of your nodes any time you want.

The output from the commands defined within the configuration file is not written to a local file. They are sent using a transport to your archive server - admittedly you could configure a transport on localhost. On the other side of the coin, when you run adhoc commands from the command line they are written to a local file and don't get sent to your

archive server. This decision was made so you don't get every user running this tool making a mess of your config archive with every single different adhoc command they want to run from the command line.

Normal users shouldn't be allowed to modify your configuration file so the administrator has complete control over what outputs are sent to the archive server on a regular basis.

-r '[type]' This option generates a node summar report of all the nodes which match the criteria. You have the ability to pass an optional "type" parameter which can be one of the following:

byproxy	This is the default report output type and will sort and group the nodes into their different proxy servers and then group by tag.
bytag	This output looks the same as "byproxy", except that it will group by tag and then by proxy.
raw	This format allows you to export the data so you are able to look at the raw inventory in Excel. The format is comma separated.
detail	This format allows you to see all the relevant data associated with a node in ns4. It outputs all the data that has been input into the configuration file.

If you run ns4 without specifying any "type" field then we get the default format which gives us the following:

```
ns4 v4.3.6
Configuration Management Tool
url http://www.noodles.org.uk/ns4.html
by Chris Mason <chris@noodles.org.uk>

[Proxy: ssh-proxy.mysite.org.uk] (nodes = 9)
 [Tags: CISCO, SP-LAB]
  - PE01-201 .. 172.16.0.201  /SP-LAB
  - PE02-202 .. 172.16.0.202  /SP-LAB
  - PE03-203 .. 172.16.0.203  /SP-LAB
  - PE04-204 .. 172.16.0.204  /SP-LAB
  - PE49-249 .. 172.16.0.249  /SP-LAB
  - PP01-101 .. 172.16.0.101  /SP-LAB
  - PP02-102 .. 172.16.0.102  /SP-LAB
  - PP03-103 .. 172.16.0.103  /SP-LAB
  - PP04-104 .. 172.16.0.104  /SP-LAB

[Proxy: host.jump.org -> LAB-TS] (nodes = 6)
 [Tags: ALCATEL, SDD-LAB]
  - PE-01 .... 100.100.100.1  /SDD-LAB
  - PE-02 .... 100.100.100.2  /SDD-LAB
  - PE-03 .... 100.100.100.3  /SDD-LAB
  - PE-04 .... 100.100.100.4  /SDD-LAB
  - PE-05 .... 100.100.100.5  /SDD-LAB
  - PE-06 .... 100.100.100.6  /SDD-LAB

Total Nodes: 15
```

The above output is first grouped into proxies, then grouped into tags and within each group we have the node name, address and then the optional location which will define the directory structure on the archive server. At this point some of the columns might not make much sense, but we will deal with them when we look at the configuration files.

-d '[file]' This option allows you to run ns4 and capture the output from the commands defined within the configuration file. The outputs are sent to your defined transports and this allows you to create a configuration backup archive.

There is an optional '[file]' parameter which allows you to store the outputs locally as opposed to sending to a transport. This could be useful if you wanted to send a local copy of the outputs to a customers, etc.

-c 'x[,...],y' This option allows you to run adhoc commands, but the output will be sent to a local file. Within the syntax you need to specify a command(s) (x) and a local file to output the results to (y).

-n '^PE01$' **-c** 'show clock, PE01.txt'	This will run the command on the node "PE01" and will send the output to a local file called "PE01.txt".
-a **-c** 'show clock, [Node].[Date].txt'	This will run the command on all nodes and will create dynamic local files based on the values within the brackets[1] at run-time.
-a **-c** 'show clock, all-nodes.txt'	This will run the command on all nodes and will output all the nodes to the same file. If you select multiple nodes then the output is appended to the local file.

Within the syntax you can specify as many commands as you want, but you must ensure that the last element is a local file to output the results to. An example of this could be:

-a **-c** 'show run, show diag, show environment, [Node].[Date].txt'	This will run the command on all nodes and will create dynamic local files based on the values within the brackets[1] at run-time.

[1]The values within brackets ([Node] and [Date]) are called special variables. Special variables are talked about later on, but they allow dynamic content to be substituted at run-time.

-l 'x,y' This option allows you to run an adhoc list of commands and the output will be appended to a local file. Within the syntax you need to specify a list file (x) and a local file to output the results to (y).

In the below example of a list file we are going to assume we are doing a health check of a Cisco router before an upgrade so it contains some commands which you might use:

```
show version
show interfaces description
show running-config
show mpls forwarding-table
```

In the examples below we are going to assume the above list is contained within a file called "cmds.txt":

-n '^PE01$' **-l** 'cmds.txt, PE01.txt'	This will run the commands within "cmds.txt" on the node "PE01" and will send the output to a local file called "PE01.txt". The output from the different commands will be appended to "PE01.txt".
-a **-l** 'cmds.txt, [Node].[Date].txt'	This will run the commands within "cmds.txt" on all nodes and will create dynamic local files based on the values within the brackets[1] at run-time. This will generate a separate local file for each node which will have all the commands within "cmds.txt" appended to each file.
-a **-l** 'cmds.txt, all-nodes.txt'	This will run the commands within "cmds.txt" on all nodes and will output all the nodes with all the

commands to the same file. If you select multiple nodes then the output is appended to the local file.

ns4 also supports entering sub-modes so we are able to enter different modes of operation to allow us to collect the output from different commands without the need of a script. If you wish to do this then you can use the braces syntax demonstrated below to specify the command and the new prompt that will be used (it redefines the default prompt for all subsequent commands until it is redefined again). If we omit the prompt then we will use the default node prompt.

```
show running-config
{admin, ^.*[Node]\(admin\)#}
show running-config (admin show running-config)
{exit}
```

In the above example we also have the ability to set an alias for the command in case we experience the example above when the command is the same within both modes. This will allow us to include the command alias within the output file.

Note: The prompt within the above syntax for sub-modes is treated as a regular expression so you need to ensure that it has been escaped if you are using characters which could potentially be treated as regular expressions that you wouldn't want to.

Note: If you do enter a sub-mode then you need to ensure that you leave the sub-mode before ns4 attempts to finish otherwise it will not see the correct prompt that it expects.

[1]The "-l" argument also supports special variables like the "-c" argument. Please see the "-c" argument for brief information on special variables.

-x This option tells ns4 to login to a node to verify that the login credentials work, etc. It won't execute any commands on a node – the only exception being commands which are defined within the "pre" and "post" statements of cartridges.

This options can be used when you are adding new nodes to the ns4 configuration and you don't want to mask our all the commands you have defined within the configuration.

-p '[logfile]' This is a new addition to ns4 which some people may say doesn't really fit in with the premise of ns4. However, due to the framework that ns4 provides it makes it very simple to implement. This feature only works for nodes which are directly connected as you are unable to proxy SNMP requests – ns4 will check that a node doesn't have a proxy defined.

Within the container options you then define the "snmp_community" and "snmp_oids" attributes which are used when polling the devices. Optionally you can specify a "logfile" which is used to store the results to, as well as the screen.

In the snippet below is an example of specifying a couple of SNMP OIDs to monitor free memory and CPU utilisation of a router:

```
---
container:
 node:
  - {id: "R1", address: "172.16.0.101"}
 options:
  snmp_oids:
   - {id: "Free Mem", oid: "1.3.6.1.4.1.9.9.48.1.1.1.6.1"}
   - {id: "CPU", oid: "1.3.6.1.4.1.9.9.109.1.1.1.1.7.1"}
  snmp_community: "CISCO"
```

-s 'x[,a=x,...]' This option allows you to run a custom script on all the nodes which match the criteria. This is a very powerful option and you can do really bad things if you get this wrong. Please test all scripts in a lab environment before you attempt to run them on a production network.

There is a separate section later on which deals with scripts in quite a lot of detail, but at the moment all you need to know is that you can pass variables into scripts from the command line. You would normally specify the name of the script file (x) and then you can pass numerous script variables using the syntax "variable_name=value".

The only part of this program which requires you to know a programming language is scripts. However, this program allows you to do an awful lot without ever touching scripts. If you wish to write your own scripts then a basic knowledge of Perl is recommended.

-n '^PE01$' **-s** 'script.nss'	This would run the script "script.nss" on the node "PE01".
-n '^PE' **-s** 'script.nss,mtu=1500'	This would run the script "script.nss" on all nodes which start with the letters "PE". It would also pass through a script variable called "mtu" which you could access through your script.
-a **-s** 'script.nss,protocol=bfd'	This is a potentially very dangerous syntax as you are running a script on all nodes within the configuration file. Be aware that it is very unlikely that you can write a generic script which can be run on multiple different nodes made by different vendors.

4.3. Options

Once you have specified the criteria and action arguments you can then specify some optional options.

-C 'file' By default the configuration file is "/etc/ns4/ns4.yaml", but if you wish to define a different configuration file for each of your different networks then you can specify an alternative one on the command line. This option takes the highest precedence against ns4 defaults and environment variables.

-R 'file' By default the cartridges file is "/etc/ns4/cartridges.yaml", but if you wish to define a different cartridges file then you can specify an alternative one on the command line. This option takes the highest precedence against ns4 defaults and environment variables.

-f '[file]' If you specify this option then ns4 will create a file in the current directory which contains a list of the nodes which failed. Once this file has been created it can then be passed back to ns4 using the "-N" parameter to troubleshoot the nodes to determine why they failed. If you don't specify an optional file then ns4 will use "ns4.[PID].failed" in the current directory, but you do have the option to overwrite this with a specific file.

-A Within the configuration file you can define alerts which allow you to define different methods of informing people about failed nodes. Defining alerts within the configuration file will be dealt with in the forthcoming sections on the configuration files.

You would usually use this option if you ran ns4 from a crontab backing up node configurations on a daily basis. Not only is it useful to know if a config backup failed, but it can also alert you to connectivity issues with nodes in your network which could be affecting customers.

-V This option is used to output version information about Perl and the different versions of modules which you have loaded that ns4 depends on. This option can be used on it's own without any critieria or action parameters and is used to output information which should be

used when reporting bugs.

-v This option is more of a troubleshooting option than one you would use regularly. You can specify this option more than once to increase the verbosity level. There are two main verbosity levels at the moment:

1
This will show you informative messages about what ns4 is currently doing. Within the current version it tells you when it is connecting to proxies and nodes and it also outputs the last line that didn't match on login/command/script timeouts.

2
This will show you what commands ns4 is executing on what nodes – even if they are being executed through scripts.

3
This will show you what ns4 is doing and what responses it is getting back. If you are having issues logging into a node then this could tell you where it is failing. Be careful when running this command when you have selected multiple nodes as the output could get quite messy and intertwined.

4
This will go one step further from 3 above and will include the internal Expect logic used to match strings so you can see where it is actually failing.

4.4. Environment Variables

ns4 supports the use of environment variables to enable the user to change some defaults which are specified within the ns4 program. This can be useful if you deploy ns4 into a user environment where you don't have root access and you don't want to keep on specifying the "-C" and "-R" command line options every time you run ns4:

NS4_CFG_FILE This allows you to overwrite the default ns4 configuration file of "/etc/ns4/ns4.yaml" with the contents of this environment variable. You can still use the "-C" command line option to overwrite this value.

NS4_CARTRIDGES_FILE This allows you to overwrite the default ns4 cartridges file of "/etc/ns4/cartridges.yaml" with the contents of this environment variable. You can still use the "-R" command line option to overwrite this value.

NS4_AI When logging commands to a syslog server using the "syslog_facility" global option the current user that ns4 is running under is also written to syslog to provide accountability of what users run what commands on your network. If using ns4 in a web environment where ns4 is launched from a CGI script the actual user will be hidden as ns4 will be run as the same user as the CGI script – this user is usually "www-data". By setting this environment variable just before you launch ns4 allows you to pass additional information that is written to syslog to allow you to identify users.

Note: Environment variables are ignored if they don't contain any textual value and the ns4 defaults will be used.

5. Special Variables

We have looked at special variables briefly during our look at the command line arguments, but as the next few sections use special variables it is probably about time we covered them in detail. Special variables allow dynamic content to be used at run-time so we can do something based upon the node which we are currently dealing with.

Special variables are defined through the use of brackets [] and will be substituted for the literal text. This is the standard format on the command line and within cartridges, but they are referred to slightly differently in scripts. In scripts they use the "dvar" method of the ScriptObject and the brackets are then omitted - but the same format is used that would have been used within the brackets. This will come apparent when we look at scripts in later sections.

5.1. Standard Variables

These are variables which are defined within the core ns4 program as they are not vendor specific. These allow users to access information which has been specified in the configuration file:

[Node]	This is the current name of the node (i.e. PE01)
[PID]	This is the current parent PID of ns4
[Proxy]	This is the current name of the proxy (i.e. ssh-proxy)
[Date]	This is the current date in the format of YYYYMMDD (i.e. 20080122)
[Time]	This is the current time in the format of HHMMSS (i.e. 170322)
[P:Username]	This is the current username of the proxy (i.e. guest)
[P:Address]	This is the current address of the proxy (i.e. 172.16.0.101)
[N:Username]	This is the current username of the node (i.e. guest)
[N:Address]	This is the current address of the node (i.e. 172.16.0.101)
[N:Prompt]	This is the current prompt of the node (i.e. [Node]#)
[N:Tags]	This contains a comma separated list of all the tags associated with the node (i.e. CISCO)
[N:Location]	This contains the "location" that is defined within the configuration file

5.2. Custom Variables

Custom variables, also known as Custom Container Options allow you to specify vendor specific information within the configuration file without having to place vendor specific code within ns4. The enable password on a Cisco router is a prime example of a custom variable:

[C:*]	You would substitute the * for a valid custom container option.

If you defined a customer container option called "enable" within the configuration file, then you could use that value within the cartridge by using the syntax "[C:Enable]".

Note: If you attempt to use a custom variable which doesn't exist then no substitution will occur.

5.3. Manipulating Special Variables

These are special variables which allow you to manipulate the contents of special variables. There are four of these variables which allow you to match contents at the beginning or at the end.

[MATCH_L*,X,C] **[MATCH_L?,X,C]**	Within this syntax the X is replaced with the name of a special variable that you wish to manipulate. The C is a single character that will specify where the match stops and the '*' or '?' indicates whether it is a greedy or standard match. To demonstrate this we can use a node of 'PE_001_RP1' and then see what happens with the following examples:

```
[MATCH_L*,Node,_]
```

In the above example this will return all the characters within the special variable [Node] that are before the last '_'. So, if we use our example above then we are going to swap this for 'PE_001'.

```
[MATCH_L?,Node,_]
```

In the above example this will return all the characters within the special variable [Node] that are before the first '_'. So, if we use our example above then we are going to swap this for 'PE'.

Note: If the character 'C' is not detected within the string then it will return the entire string.

[MATCH_R*,X,C]
[MATCH_R?,X,C]

This command uses the same syntax from the above example, but instead of matching from the left, we use the same theory but match from the right.

6. Configuration Files

ns4 uses two different configuration files: "cartridges.yaml" and "ns4.yaml". I have picked YAML, a data-centric mark-up language for the format of the configuration files because it is best suited for hierarchical configuration files which fits our requirements nicely. I did originally start off with XML, but its bloat soon become apparent when it started to affect loading times of ns4, so I decided to move to a more lightweight approach.

I am not going to go into the details of YAML, as I intend that you learn how to write configuration files for ns4 as opposed to learning how to write YAML. I use an extremely basic subset of the YAML language and don't touch some of the more complicated constructs, but if you wish to learn it in more detail then you can find more information at http://yaml.org/spec/current.html. We currently only support version 1.0 of the YAML specification as it covers everything which we need.

YAML uses indentation to denote hierarchy, but be aware that indentation is done by the use of spaces and not tabs - tabs are strictly prohibited in YAML. This is something to be aware of if your editor doesn't automatically convert tabs into spaces when it saves your file.

As with most modern languages there are multiple different ways to write the same thing, YAML is no exception. It supports a compressed view as well as an expanded view of data or a mixture of both. Throughout this guide I have ensured that I have used the fully expanded format of YAML, to be consistent, but there are shortcuts which can be used to create a more compact view.

Note: The YAML specification defines that all YAML files (including "cartridges.yaml" and "ns4.yaml") must start with the "---" sequence on a single line to denote the beginning of the file.

6.1. cartridges.yaml

Based on my early requirements of multi-vendor I needed a way to define cartridges that allowed me to define rules about how to login to different pieces of kit. It was important that people could write cartridges who didn't know how to program and I needed a modular approach so I could keep them isolated from the main program. By using modular cartridges this allowed ns4 to be completely multi-vendor as there was no vendor specific logic found within the main program.

Every node within the configuration file must be associated with a cartridge - it is like a label which tells the program what sort of node it is and how to login/logout from it.

The excerpt below is the cartridge schema written in YAML. It defines the hierarchical rules about what elements can exist where within a cartridge. Within YAML, the hyphen "-" denotes that it is a list of objects following (i.e. the hyphen under the cartridge indicates that we are expecting one or more cartridges below). The hyphen can be omitted if there is only a single element within the list.

```
---
cartridge: MANDATORY
 -
  id: MANDATORY
  prompt: MANDATORY
   expect: OPTIONAL
    -
     value: MANDATORY
     send: MANDATORY
   value: DEPENDS
  pre: OPTIONAL
   -
    expect: OPTIONAL
     -
      value: MANDATORY
      send: MANDATORY
    prompt: OPTIONAL
    value: DEPENDS
  post: OPTIONAL
```

```
   -
    expect: OPTIONAL
     -
      value: MANDATORY
      send: MANDATORY
    prompt: OPTIONAL
    value: DEPENDS
  options: OPTIONAL
   tag: OPTIONAL
    -
   terminator: OPTIONAL
   pre_login_seq: OPTIONAL
   logout_cmd: OPTIONAL
   rawpty: OPTIONAL
   alt_username_prompt: OPTIONAL
   alt_password_prompt: OPTIONAL
```

I am going to start by creating a basic cartridge for us to use for Cisco routers and then we are going to extend this cartridge as we explain more from the schema. The basic concept of a cartridge is to define the prompt that you expect to see when you login to a node. Within a cartridge there are two mandatory elements: *id* and *prompt*.

```
---
cartridge:
 id: cisco-ios
 prompt: "^.*[Node].*>\\s*"
```

Note: The value of *prompt* is a regular expression and we support the use of Perl's special escape sequences (i.e. \w, \W, \s, \S, \d, \D, etc), but they need to be escaped so the meaning can reach the interpreter and can be interpreted correctly.

Now ns4 is able to determine when it has successfully logged into a node as it knows what prompt it is expecting. You will see the use of special variables in the above example which allows us to customise the prompt for every different node we login to. Normally, once we are logged into a node we would run through the commands defined within the configuration file then we would exit. When we exit from a node we would expect to see the prompt we have defined within our proxy configuration to know we have logged out from the node successfully.

As well as the *id* and *prompt* elements, cartridges also support *pre* and *post* elements. The *pre* element allows us to run commands defined within the cartridge before we declare that we are logged into a node and give control back to ns4. The *post* element allows us to run commands before we logout from a node, or it can be used to actually logout from a node if the logout procedure is slightly different from what ns4 does by default.

Normally, when you run a command on a Cisco router it uses a pager which pauses at the end of every page. This doesn't really work that well on non-interactive scripts so there is a command which we can type that disables this: "terminal length o". We can use the *pre* element to execute this command before control is given back:

```
---
cartridge:
 id: cisco-ios
 prompt: "^.*[Node].*>\\s*"
 pre: terminal length 0
```

Note: When we see the word "DEPENDS" within the YAML schema it usually relates to the *value* element. Within YAML, if you are only defining a single value within an element then that value will get assigned to the *value* element. If you define multiple elements within an element then you need to specially define the *value* element. There are two special elements which can coexist alongside the *value* element within the cartridge configuration file: *expect* and *prompt*.

Another way of writing the above example to explain this is:

```
---
cartridge:
 id:
  value: cisco-ios
 prompt:
  value: "^.*[Node].*>\\s*"
 pre:
  value: terminal length 0
```

The above example doesn't really help us on a Cisco router if we need to execute commands from privileged mode (i.e. enable mode). By default you are put into un-privileged mode and you need to execute the "enable" command before you enter privileged mode. The two different modes are differentiated through the prompt (in privileged mode the prompt ends with a "#" as opposed to a ">" in un-privileged mode).

As there is no vendor specific logic within ns4 it doesn't know what privileged mode is or an enable password. This means we have to define one using custom container options within the configuration file and you have to program the cartridge yourself. There are generally multiple different ways to do the same thing in life, and this is no exception. We can either use the *prompt* element, or we can extend the *pre* element to support this. They have a very subtle difference - the *prompt* element will do something based upon the prompt it finds, but the *pre* element will do it regardless of the prompt. In this scenario we only want to attempt to enter privileged mode if we are not already in it so the *prompt* element works well, however, we are going to extend the *pre* element to start with.

We have this idea of an *expect* element which allows us to do something based on what we find. It allows us to put conditional logic into cartridges and is what makes them so powerful and flexible. Within the schema above the *pre* element is defined as a list, so we can support multiple occurrences:

```
---
cartridge:
 id: cisco-ios
 prompt: "^.*[Node].*>\\s*"
 pre:
  -
   terminal length 0
  -
   value: enable
   expect:
    value: "Password:"
    send: "[C:Enable]"
   prompt: "^.*[Node].*#\\s*"
```

Note: If we have a blank send element then we will send a single CR or whatever the default terminator is.

Both the *pre* and *post* elements support a *prompt* element, but this *prompt* element is slightly different from the main *prompt* element at the top level of the cartridge. The *prompt* element within the *pre* and *post* elements allows us to rewrite the prompt as we go along, so every time we hit a *prompt* element we change what ns4 expects the prompt to be for the next action. The main *prompt* element at the top level specifies what we should initially get when we login to the node.

So, we start off by sending through "terminal length 0" and we expect the prompt to still be the same prompt we defined at the beginning (i.e. "^.*[Node].*>\\s*"). We then send through "enable" and we are expecting the prompt to be "^.*[Node].*#\\s*", however, we know that we are going to be prompted for a password and if we get that prompt we are going to send "[C:Enable]" which is the enable password defined within the custom container options (explained later). If the password is accepted then the prompt is going to match and ns4 will set the internal prompt it expects, to be the privileged version.

However, this cartridge has quite a serious flaw - you can login to a Cisco router and be put automatically into privileged mode. If this happens then you would never match the top level *prompt* element of "^.*[Node].*>\\s*" and it would time out waiting for this prompt. This is where we can use the *expect* element within the *prompt* element to allow us to work regardless of the mode you login as:

```
---
cartridge:
 id: cisco-ios
 prompt:
  expect:
   -
    value: "^.*[Node].*>\\s*"
    send: enable
   -
    value: "Password:"
    send: "[C:Enable]"
  value: "^.*[Node].*#\\s*"
 pre: terminal length 0
```

In the example above we define the value of the prompt to be what we expect to see (i.e. the privileged prompt) and we can define *expect* elements in case we find something else. This works slightly different to the *pre* element as it allows us to do something before we hit the initial prompt, whereas the *pre* element must match the initial prompt first. The *expect* elements are optional elements which define what to send if we see the text which they refer to.

So, if we come across an un-privileged prompt then we are going to send the command "enable". When we receive the prompt for the password we are going to send the enable password. Hopefully at the end of this we should get the correct prompt and then we are going to move onto the *pre* element.

The *pre* element has a partner which is the *post* element which works when you are logging off from a node. This can be used to dismiss messages asking you to save configs or messages asking if you are sure you want to quit. As our Cisco routers don't generally prompt when you logoff we are going to assume it asks if you are sure you want to quit:

```
---
cartridge:
 id: cisco-ios
 prompt:
  expect:
   -
    value: "^.*[Node].*>\\s*"
    send: enable
   -
    value: "Password:"
    send: "[C:Enable]"
  value: "^.*[Node].*#\\s*"
 pre: terminal length 0
 post:
  value: exit
  expect:
   value: "are you sure"
   send: yes
```

The *post* element is going to cause ns4 to send "exit", however we have omitted the *prompt* element because a little bit of magic happens here. ns4 knows that the last element within a *post* element needs to logout from the node, this means it knows that the next prompt it should receive will be the proxies' prompt, or an EOF when we aren't using a proxy. In the above example, if we are prompted with the text "are you sure" then we are going to send "yes". The above is a very simple example, but it works exactly the same as the *pre* element.

Note: If you specify the *post* element, then ns4 assumes you are going to deal with logging off from the node and will not send the logout sequence, even if it is specified through the *logout_cmd* element. You will need to ensure that the last element within a *post* element is the logout sequence and the *prompt* element will be ignored if you provide it. As stated above, ns4 will automatically determine what prompt it expects.

Note: The value you specify within the *pre* and *post* element will be treated as literal values, with the exception of a special escape sequence. The sequence "\cC", when specified as the complete *value* element will disconnect the

session and report a successful logout. This sequence is for terminal server support which will generally require you to send an escape sequence to be able to disconnect from the line:

```
id: cisco-ios-via-ts
pre: terminal length 0
prompt:
 expect:
  - {value: "console disabled", send: "\cC"}
  - {value: "^.*[Node].*>\\s*", send: "enable"}
  - {value: "Password:", send: "[C:Enable]"}
 value: "^.*[Node].*#\\s*"
post:
 expect: {value: "\r\n\r\n\r\n", send: "\cC"}
 value: exit
options:
 pre_login_seq: "\r"
```

Within the above configuration you will see an example of implementing a cartridge that allows you to connect through a terminal server. When you are connecting to nodes through a terminal server it is advisable to use the null proxy - if your sessions go through a proxy to reach a terminal server then ns4 will need to reconnect through the proxy for every session.

Although my examples are not overly complex, I hope they do the job. As you can see, when you use the *pre*, *post* and *prompt* elements with the *expect* element you could potentially log into any node regardless of what it throws at you!

Finally we have the cartridge options which are listed below:

pre_login_seq

This element has been introduced for terminal server support. This enables you to connect to a node which is connected to a line of a terminal server. The reason for this command is to be able to "wake up" the line to let the terminal server know there is someone connected to the line. Normally when you connect to a node behind a terminal server you need to send an initial carriage return before you are connected to the device. The *pre_login_seq* element allows you to specify what character is initially sent.

alt_username_prompt

By default ns4 will look for a specific prompt to identify the username prompt. If the node has a custom login prompt then we may specify a regex within this element which will be used instead of the default regex. As with the *prompt* element you need to ensure that you escape all special characters.

alt_password_prompt

By default ns4 will look for a specific prompt to identify the password prompt. If the node has a custom password prompt then we may specify a regex within this element which will be used instead of the default regex. As with the *prompt* element you need to ensure that you escape all special characters.

rawpty

This element allows you to specify whether the node supports disabling of local echo. By default ns4 will attempt to disable local echo as it makes the transfer more pipelined, but it is unaware if it is successful or not so it needs to be told if a node supports disabling of local echo. By default it assumes that no nodes support disabling of local echo which is why you need to specifically state whether it does support it. If you don't specify a node supports disabling of local echo and it does, then you will find the first line of your output missing. The *rawpty* element within the cartridge cannot be used to disable ns4 attempting to disable local echo (except for null proxy - explained below), but the *rawpty* element within the proxy can be used to disable it as ns4 attempts to disable local echo on a per-connection basis.

If we decide to access our node without going through a proxy (null proxy) then the *rawpty* element within the cartridge will not attempt to disable local echo if you

explicitly set this value to "no".

terminator (default: "LF") — This element controls the end-of-line termination character that ns4 sends when it is sending commands to a node. By default ns4 will use LF (\n) as the default line terminator, but some nodes require CRLF (\r\n). You can specify as many variants of the terms CR or LF within this value.

logout_cmd (default: "exit") — This element allows you to tell ns4 to send a separate exit sequence as opposed to "exit". This allows you to send "logout" as an alternative if the node supports this. However the use of the *post* element will overwrite the exit sequence and ns4 expects you to logout from the node using the *post* element if supplied.

[tag] — This element works the same as the tag element within the node options. Any tags which you specify within this element are combined with the tag elements within the node options. This allows you to set tags related to different vendors. You can select all the Cisco routers, or switches based on tags without having to define them within the node options.

Note: The brackets [] indicate that this element supports a list to specify multiple entries.

6.2. ns4.yaml

Compared to cartridges, the main configuration file is relatively easy. The only slightly complex difference between cartridges and the main configuration file is that the latter supports hierarchical inheritance. Nodes have to be defined within containers and containers can be defined within containers to provide inheritance. The schema for the main configuration file is as follows:

```
---
syslog_facility: OPTIONAL
gpg_path: OPTIONAL
terminator: OPTIONAL
default_report_type: OPTIONAL
explicit_null_tag: OPTIONAL
min_nps_null_proxy: OPTIONAL
max_sessions_null_proxy: OPTIONAL
socks_wrapper: OPTIONAL
max_buffer_kb: OPTIONAL

proxy: MANDATORY
 -
  id: MANDATORY
  address: OPTIONAL
  username: MANDATORY
   ask: OPTIONAL
   type: DEPENDS
  password: MANDATORY
   encoding: OPTIONAL
   value: DEPENDS
   ask: OPTIONAL
   type: DEPENDS
  method: MANDATORY
  prompt: MANDATORY
  socks_wrapper: OPTIONAL
  local_socks_port: OPTIONAL
  terminal: OPTIONAL
  terminator: OPTIONAL
  logout_cmd: OPTIONAL
  rawpty: OPTIONAL
  tag: OPTIONAL
   -
  max_sessions: OPTIONAL
  min_nodes_per_session: OPTIONAL
  max_proxy_retries: OPTIONAL
  alt_username_prompt: OPTIONAL
  alt_password_prompt: OPTIONAL
```

```
  timeout: OPTIONAL

container: MANDATORY
 -
  container: OPTIONAL
   -
  node: MANDATORY
   -
    id: MANDATORY
    address: OPTIONAL
  options: MANDATORY
   cartridge: MANDATORY
   username: MANDATORY
    ask: OPTIONAL
    type: DEPENDS
   password: MANDATORY
    encoding: OPTIONAL
    value: DEPENDS
    ask: OPTIONAL
    type: DEPENDS
   method: MANDATORY
   alert: OPTIONAL
   transport: OPTIONAL
   proxy: OPTIONAL
    -
   location: OPTIONAL
   command: OPTIONAL
    -
     alias: OPTIONAL
     value: DEPENDS
   tag: OPTIONAL
    -
   custom: OPTIONAL
    ...
     encoding: OPTIONAL
     value: DEPENDS
     ask: OPTIONAL
     type: DEPENDS
   gpg_keyid: OPTIONAL
    -
   max_node_retries: OPTIONAL
   timeout: OPTIONAL
   snmp_community: OPTIONAL
   snmp_oids: OPTIONAL
    -
     id: MANDATORY
     oid: MANDATORY

transport_ftp: OPTIONAL
 -
  id: MANDATORY
  address: MANDATORY
  username: MANDATORY
  password: MANDATORY
   encoding: OPTIONAL
   value: DEPENDS
  max_retries: OPTIONAL

transport_sftp: OPTIONAL
 -
  id: MANDATORY
  address: MANDATORY
  username: MANDATORY
  password: MANDATORY
   encoding: OPTIONAL
   value: DEPENDS
  max_retries: OPTIONAL

alert_smtp: OPTIONAL
 -
  id: MANDATORY
  relay: MANDATORY
  username: OPTIONAL
  password: OPTIONAL
```

```
   from: MANDATORY
   to: MANDATORY
    -
   cc: OPTIONAL
    -
   timeout: OPTIONAL

alert_http: OPTIONAL
 -
   id: MANDATORY
   post_url: MANDATORY
   proxy_url: OPTIONAL
   timeout: OPTIONAL
```

Note: When we see the word "DEPENDS" within the YAML schema it usually relates to the *value* element. Within YAML, if you are only defining a single value within an element then that value will get assigned to the *value* element. If you define multiple elements within an element then you need to specially define the *value* element. There are two special elements which can coexist alongside the *value* element within the main configuration file: *encoding* and *alias*.

The hierarchical inheritance model allows container options defined within a higher layer to be used within a child container. Container options defined within a lower layer overwrite previously defined values in a higher layer as opposed to being merged.

```
---
container:
 container:
  - # Child Container 1
   node:
    id: PE01
    address: 172.16.0.101
  - # Child Container 2
   node:
    id: PE06
    address: 172.16.0.106
   options:
    command: show version
 options:
  command:
   -
    show running-config
   -
    show startup-config
  tag: PE
```

In the above example only the *command* element is overwritten for PE06 and the *tag* element is inherited down. PE01 will use the commands defined within the top container, but PE06 re-defines those commands. I hope the above example has explained the hierarchical inheritance model that ns4 adopts. If you are still unsure about how it works then hopefully the examples in the next few sections will help to explain it further.

6.2.1. Element Encoding

Some elements support the use of the *encoding* element which allows you to encode the value within the configuration file. Within the current version of ns4 the only element which allows the *encoding* element is the *password* element and any element which is a custom container option.

At the moment ns4 supports two different encoding schemes: "rot13" and "base64". For "rot13" there is a webpage at http://www.rot13.com which allows you to encode values. For "base64" encoding you need to have the MIME::Base64 module installed, however, I believe this is part of the core Perl distribution. Encoding is not encryption and is there to stop people with prying eyes looking over your shoulder when you are working on a configuration file. You should configure your Unix file permissions so that only people who have access to ns4 will have access to the configuration files. As a general rule you should only give access to ns4 to people who already have access to your data network.

The following examples demonstrate the difference between encoding and not encoding:

```
---
container:
 options:
  password: secret
```

```
---
container:
 options:
  password:
   encoding: rot13
   value: frperg
```

```
---
container:
 options:
  password:
   encoding: base64
   value: c2VjcmV0
```

For the Perl people among us we could use the following Perl snippet to implement a simple rot13 encoder/decoder:

```
tr/a-zA-Z/n-za-mN-ZA-M/
```

For Base64 encoding we have the MIME::Base64 module to assist us:

```
perl -MMIME::Base64 -e 'print encode_base64("<PASSWORD>");'
```

6.2.2. Ask Attribute

Some elements support the "ask" attribute which tells ns4 to prompt the user when it runs to determine what value to use for this attribute. At the moment this is supported on "username", "password" attributes and custom container options. When a user runs ns4 they need to have access to the configuration file to access the login details for nodes – this could be a security risk for certain people – even though only people with access to the nodes should be allowed to use ns4.

This option allows us to remove the sensitive information from within the configuration file and prompt the user when they run ns4. ns4 will only ask for the information once per item and will only ask for it if it actually needs it – i.e. if you are running the report it won't ask for it and if you only select a single node then it will only ask you for the information that it needs to connect to that node.

As an example let us prompt the user for the username and password when connecting to a proxy – this would be achieved through the following configuration:

```
---
proxy:
 id: "nms.myisp.org"
 username: {ask: "nms.myisp.org Username"}
 password: {ask: "nms.myisp.org Password", type: "password"}
```

As well as the "ask" attribute we also support the optional "type" attribute only if it precedes an "ask" attribute. The only valid value for the "type" attribute is "password" at the moment and it is used to turn off local echo when ns4 prompts for this value.

The value you specify after the "ask" keyword is the text that ns4 will prompt the user with and is also used as the key for ns4 when storing the value. If you used that same text in another "ask" attribute for a different proxy then ns4 will use the same value and won't prompt the user for the same value twice.

```
ns4 v4.3.6
Configuration Management Tool
url http://www.noodles.org.uk/ns4.html
by Chris Mason <chris@noodles.org.uk>

nms.myisp.org Password:
nms.myisp.org Username:
```

As we can see from the above output it has asked for the password before the username which some people may object to. By default ns4 will sort the values based on the text and ask for them in that order. To get around this limitation there is a special syntax which we can use to force ns4 to ask in a different order:

```
---
proxy:
 id: "nms.myisp.org"
 username: {ask: "1|nms.myisp.org Username"}
 password: {ask: "2|nms.myisp.org Password", type: "password"}
```

In the above example we have added a number followed by the pipe "|" sign to the beginning of the text. This won't actually be used in the text output, but ns4 will use this as a sorting key – you can actually put anything before the pipe, and if the pipe is omitted then the text is used as the sorting key. If we run ns4 again we can now see our desired effect:

```
ns4 v4.3.6
Configuration Management Tool
url http://www.noodles.org.uk/ns4.html
by Chris Mason <chris@noodles.org.uk>

nms.myisp.org Username:
nms.myisp.org Password:
```

6.2.3. Global Options

Global options are defined at the root of the configuration file. These allow you to control things which are not really specific to anything in particular, but are there to control ns4 core functions.

syslog_facility — By setting this option to a valid SYSLOG facility, ns4 will send auditing information for all commands which are run – even through scripts, to the local syslog server using the Perl module Sys::Syslog.

gpg_path — This allows you to specify the path to the gpg binary if you have GnuPG installed on your system. This option is mandatory if you are using the "gpg_keyid" for a node. This allows you to encrypt the output before sending it via ftp with a GnuPG public key for security.

terminator (default: "LF/CRLF") — The terminator element controls the end-of-line termination character that ns4 uses when it is writing local files. Local files are generated as a result of the various action command line options. Under Unix environments ns4 will use LF (\n) as the default line

terminator. When running under Cygwin on Windows ns4 will use CRLF (\r\n). You can specify as many variants of the terms CR or LF within this value.

default_report_type (default: "byproxy") — This allows you to change the default report type without having to keep specifying a different type when using the "-r" parameter. This option takes the same values which are valid for the "-r" parameter mentioned earlier.

explicit_null_tag — If this option exists, regardless of what it is set to, then it will enable the use of a "NULL" tag on all nodes which use the null proxy. As we have proxy implied tags, this allows us to also tag the null proxy nodes.

min_nps_null_proxy (default: 1) — This controls the loading algorithm and determines when to spawn new children. This option is identical to "min_nodes_per_session" which is defined on the proxy, but this option is relevant to sessions which use the null proxy option. The defaults for this value are slightly lower than when using a proxy and they provide a speed benefit. The default values work very well and are generally left alone.

max_sessions_null_proxy (default: 12) — This controls the amount of sessions that ns4 will spawn when using the null proxy. This option is identical to "max_sessions" which is defined on the proxy, but this option is relevant to sessions which use the null proxy option. The defaults for this value are slightly higher than when using a proxy and they provide a speed benefit. The default values work very well and are generally left alone.

max_buffer_kb (default: unlimited) — This option allows you to specify the maximum amount of data in KB that ns4 will attempt to retrieve from a node per command. In some circumstances the output from a command can be quite considerable (i.e. "show tech-support" on a Cisco router) and it can be useful to limit it. This will cause ns4 to stop as soon as this value is exceeded.

socks_wrapper — Please see section 7 – "Socks Wrapper"

6.2.3. Proxies

A proxy defines a device which you can hop through to be able to connect to the node. We also support the ability to hop through multiple proxies to reach your node. In the stereotypical Service Provider's network you would have your network devices situated within a different security zone to your desktop PCs. You would have an NMS DMZ which contains various proxy servers that limit access to your network devices as not everyone within your desktop environment should have access to them.

When you login to a device you would normally require Telnet or SSH access (telnet is a serious security risk as passwords are sent in clear text and should not be permitted). Following our above model you would SSH to your NMS proxy and then you SSH to your nodes.

Proxies have various elements to control how ns4 works and I am hoping that most of them look obvious, but just in-case they are explained below:

id — This is the id of the proxy which is used to allow container options to reference the proxy.

address[:port] — This is the IP address of the proxy server. However, this doesn't necessarily need to be an IP address and a hostname will also be accepted. You can also specify an optional port if you wish to change the default.

This value is optional and if omitted ns4 will use the "id" for the address. It will also output "DNS" as the address in report outputs.

username — This is the username for the proxy server.

password This is the password for the proxy server.

method This is a string which determines what command is going to be run to connect to the proxy. It will consist of the binary (i.e. telnet/ssh) and the parameters to pass to it. We allow three different dynamic variables within brackets which can be used to construct this string: [Host], [Port], [User] (case insensitive).

This string is passed directly to the Unix "exec" function call, so you need to ensure that you place a comma between parameters. I am hoping that the following examples will demonstrate this:

"/usr/bin/telnet [host], [port]"
This will connect by running "/usr/bin/telnet" and it will substitute the [host] and [port] parameters for the real values at runtime.

"/usr/bin/ssh -p [port], [user]@[host]"
This will connect by running "/usr/bin/ssh" and it will substitute the [host], [port] and [user] parameters for the real values at runtime.

"/usr/bin/ssh -v, -p [port], -l[user], [host]"
This is an extension to the example above and demonstrates the flexibility in being able to enable ssh to be more vocal for you to diagnose any connection related issues. Each parameter needs to be separated by a comma.

Note: So ns4 is able to work out default port numbers the binary at the left of the string must contain either "ssh" or "telnet".

prompt This is the prompt that you expect to see when you login to the proxy. The value is a regular expression and we support the use of Perl's special escape sequences (i.e. \w, \W, \s, \S, \d, \D, etc), but they need to be escaped so the meaning can reach the interpreter and can be interpreted correctly.

socks_wrapper Please see section 7 – "Socks Wrapper"

local_socks_port Please see section 7 – "Socks Wrapper"

terminal This allows you to specify the terminal when you login to the proxy before you attempt to connect to any nodes. Some nodes require the terminal to be set otherwise the output can be screwed on non-interactive sessions. A good value to use for this is "vt100" which would cause the command "export TERM=vt100" to be executed on the proxy before connecting to the node.

terminator (default: "LF") The terminator element controls the end-of-line termination character that ns4 uses when it is sending commands through Expect for the proxy. By default ns4 will use LF (\n) as the default line terminator which matches Unix, but some other proxies require other terminators. You can specify as many variants of the terms CR or LF within this value.

logout_cmd (default: "exit") This allows you to specify an alternative command that Expect will send when you attempt to logout from the proxy. By default ns4 will send "exit", but some proxies may require you to use the "logout" command instead.

rawpty (default: yes) This determines if ns4 will attempt to set a raw pty (attempt to disable local echo) before it spawns the session. You can either specify "yes" or "no". If this is set on nodes which support raw pty, it will stop commands being echo'ed back to the client and also

	make command processing cleaner. As a general rule this option is left alone and the "rawpty" option within the container options is set to "yes" if the node supports it. Note: You may need to disable this option if you use a Solaris box as your proxy. From experience, SunOs 5.8 doesn't allow setting a raw pty otherwise it misinterprets "\n" when sending commands.
alt_username_prompt	By default ns4 will look for a specific prompt to identify the username prompt. If the proxy has a custom login prompt then we may specify a regex within this element which will be used instead of the default regex. As with the *prompt* element you need to ensure that you escape all special characters.
alt_password_prompt	By default ns4 will look for a specific prompt to identify the password prompt. If the poxy has a custom password prompt then we may specify a regex within this element which will be used instead of the default regex. As with the *prompt* element you need to ensure that you escape all special characters.
[tag]	This element works the same as the tag element within the node options. Any tags which you specify within this element are combined with the tag elements within the node options. This allows you to set tags related to different proxies. Note: There is a caveat relating to proxy load balancing - if you use this then the node will inherit the proxy tags from all the proxies specified, not just the one that is selected through load balancing.
max_sessions (default: 5)	This controls the amount of sessions that ns4 will spawn to the same proxy. Each child that ns4 spawns will attempt to connect to a proxy, this means you can have multiple connections to the same proxy simultaneously.
min_nodes_per_session (default: 1)	This controls the loading algorithm and determines when to spawn new children. This is used with "max_sessions" and it won't spawn another session unless the previous session has "min_nodes_per_session" defined through it. The default values work very well and are generally left alone.
max_proxy_retries (default: 3)	This defines the amount of times that ns4 will attempt to connect to a proxy server in case the connection to the proxy server times out.
timeout (default: 10 sec)	This is the timeout that ns4 will wait for an expected prompt before bombing out. The "max_proxy_retries" * "timeout" is used to determine how long ns4 will wait before determining a proxy is dead.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element. The brackets [] indicate that this element supports a list to specify multiple entries.

Taking the above proxy elements into consideration an example proxy configuration would look like:

```
---
proxy:
 id: ssh-proxy
 address: 20.30.10.1
 method: "ssh [user]@[host]"
 username: root
 password:
  encoding: rot13
  value: frperg
 prompt: "^ssh-proxy:/"
```

6.2.4. Transports

Transports are used as destinations for the output that ns4 collects from nodes. This output is generated from the list of commands within the configuration file. Only commands within the configuration file can be sent to transports and commands which are run from the command line can only be outputted locally. The standard behaviour of ns4 is to create a directory on the transport server which matches the current date using the format "YYYYMMDD" and then a directory with the name of the command (unless an alias is specified). The file is then written using the format "[Node].[Date].txt". Multiple commands within the configuration file will generate multiple directories being made on the transport server; one for each command.

Transports are not mandatory unless you define some commands under the container options. If you decided you don't want to define any commands then ns4 will login to the node and then logout from the node without doing anything. This could be useful if you wish to use ns4 for health checking purposes.

6.2.4.1. transport_ftp

The "transport_ftp" element is used to define a transport which uses an FTP server as the destination. It is defined using the following elements:

Id	This is the id of the transport.
address	This is the IP address of the ftp server. However, this doesn't necessarily need to be an IP address and a hostname will also be accepted.
username	This is the username of the ftp server.
password	This is the password of the ftp server.
max_retries (default: 3)	This determines the maximum amount of times ns4 will attempt to connect to an ftp server before declaring it dead.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element.

Taking the above transport_ftp elements into consideration an example ftp configuration would look like:

```
---
transport_ftp:
 id: db-server
 address: 20.30.10.76
 username: ns4
 password:
  encoding: rot13
  value: frperg
```

Transports are not automatically implied from the configuration for nodes and need to be specified under container options (like proxies) (as of v4.3.6 of ns4). Due to hierarchical inheritance rules, nodes will inherit container options which are defined at a higher level. This means that, by default, no transports will be used unless they are defined on the root container under options with the "transport" attribute.

Note: ns4 has mandatory and optional Perl modules which are used depending on what the user has configured. Transports fall within the category of optional Perl modules and to use the "transport_ftp" element you will need to ensure you have installed the Net::FTP module.

6.2.4.2. transport_sftp

The "transport_sftp" element is used to define a transport which uses an SFTP server as the destination. It is defined using the following elements:

id	This is the id of the transport.
address	This is the IP address of the sftp server. However, this doesn't necessarily need to be an IP address and a hostname will also be accepted.
username	This is the username of the sftp server.
password	This is the password of the sftp server.
max_retries (default: 3)	This determines the maximum amount of times ns4 will attempt to connect to a sftp server before declaring it dead.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element.

Taking the above transport_sftp elements into consideration an example sftp configuration would look like:

```
---
transport_sftp:
 id: db-server
 address: 20.30.10.76
 username: ns4
 password:
  encoding: rot13
  value: frperg
```

Transports are not automatically implied from the configuration for nodes and need to be specified under container options (like proxies) (as of v4.3.6 of ns4). Due to hierarchical inheritance rules, nodes will inherit container options which are defined at a higher level. This means that, by default, no transports will be used unless they are defined on the root container under options with the "transport" attribute.

Note: ns4 has mandatory and optional Perl modules which are used depending on what the user has configured. Transports fall within the category of optional Perl modules and to use the "transport_sftp" element you will need to ensure you have installed the Net::SSH2 module.

6.2.5. Alerts

Alerts are used to notify users via a certain method that a node has failed when ns4 attempted to connect to it. Depending on the alert type depends on what elements you are expected to fill in. To enable ns4 to send alerts you must specially enable it from the command line using the "-A" option. It is generally a good idea to enable this option if you are running ns4 from a crontab to collect your network configurations. An alert won't be sent if there weren't any nodes which couldn't be accessed and it will only include nodes which failed.

6.2.5.1. alert_smtp

The "alert_smtp" method allows you to define an alert which will send an email containing the list of failed nodes. It uses the following elements:

id	This is an id to use for a reference for this alert.
relay	This is your SMTP relay which you can use to send email through. You must specify a relay which is capable of forwarding traffic to the recipients specified. This can either be an IP address or a hostname.

from	This is the address which will be used in the "MAIL FROM" part of the SMTP message and also the "From:" part of the message headers.
[to]	This element allows you to specify multiple SMTP recipients which will receive the message.
[cc]	This element allows you to specify multiple SMTP recipients which will receive a carbon copy of the message.
username	This is used to specify a username if your SMTP relay requires authentication.
password	This is used to specify a password if your SMTP relay requires authentication.
timeout (default: 15 sec)	This is a timeout which is used when attempting to connect to the SMTP relay.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element. The brackets [] indicate that this element supports a list to specify multiple entries.

Taking the above alert elements into consideration an example alert configuration would look like:

```
---
alert_smtp:
 id: smtp-relay
 relay: 20.30.40.56
 from: root@ns4-host-box
 to:
  -
   alerts@my-company.org
  -
   sysadmin@my-company.org
```

Alerts are not automatically implied from the configuration for nodes and need to be specified under container options (like proxies) (as of v4.3.6 of ns4). Due to hierarchical inheritance rules, nodes will inherit container options which are defined at a higher level. This means that, by default, no alerts will be used unless they are defined on the root container under options with the "alert" attribute.

Note: ns4 has mandatory and optional Perl modules which are used depending on what the user has configured. Alerts fall within the category of optional Perl modules and to use the "alert_smtp" element you will need to ensure you have installed the Net::SMTP module.

6.2.5.2. alert_http

The "alert_http" method uses the HTTP POST method to post the list of failed nodes to a web page. This can be used in environments where SMTP access to an outside server is not possible but they allow HTTP access. It uses the following elements:

id	This is an id to use for a reference for this alert.
post_url	This is the complete URL address to the CGI script that you want ns4 to post the results to. It will use the following elements to post the data as:

start_time	This contains the start time of when ns4 was started.
elapsed_time	This contains the total amount of time that ns4 spent logging into nodes.

	cmd_line	This contains the command line which was passed to ns4.
	data	This contains the actual list of failed nodes.

An example of a post_url would be:
http://www.mysite.com/cgi-bin/ns4.cgi

proxy_url — If you don't have direct access to the HTTP server then you have the option of specifying a proxy URL. The format of this matches the standard Unix format for the http_proxy environment variable.

An example of a proxy_url would be:
http://username:password@proxy_address

timeout (default: 30 sec) — This is a timeout which is used when attempting to connect to the HTTP server.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element. The brackets [] indicate that this element supports a list to specify multiple entries.

Taking the above alert elements into consideration an example alert configuration would look like:

```
---
alert_http:
 id: http-server
 post_url: http://www.mysite.com/cgi-bin/ns4.cgi
```

The above example will do a HTTP POST to ns4.cgi and an example ns4.cgi is included below (the below scenario was used when there was no direct access to an SMTP server but the results still needed to be sent via email):

```
#!/usr/bin/perl -Tw
use strict;

use Net::SMTP;
use CGI::Simple;

print "Content-Type: text/html\r\n\r\n";

my $qCGI = new CGI::Simple;

if (defined $qCGI->param ("data")) {
  if (my $nSMTP = new Net::SMTP ("127.0.0.1")) {
    $nSMTP->auth ("username", "password");
    if ($nSMTP->ok) {
      $nSMTP->mail ("ns4-alerts\@myserver.com");
      if ($nSMTP->ok) {
        $nSMTP->to ("ns4-clients\@myserver.com");
        if ($nSMTP->ok) {
          $nSMTP->data;
          if ($nSMTP->ok) {
            $nSMTP->datasend ("X-Priority: 1\n");
            $nSMTP->datasend ("From: ns4-alerts\@myserver.com\n");
            $nSMTP->datasend ("To: ns4-clients\@myserver.com\n");
            $nSMTP->datasend ("Subject: " . $qCGI->param ("cmd_line") . "\n\n");
            $nSMTP->datasend ("Start Time: " . $qCGI->param ("start_time") . "\n");
            $nSMTP->datasend ("Elapsed Time: " . $qCGI->param ("elapsed_time") . "\n\n");
            $nSMTP->datasend ($qCGI->param ("data") . "\n");
            $nSMTP->dataend;
            if (not $nSMTP->ok) {
              print $nSMTP->message;
            }
            else {
              print "OK\n";
```

```
            }
          }
          else {
            print $nSMTP->message;
          }
        }
        else {
          print $nSMTP->message;
        }
      }
      else {
        print $nSMTP->message;
      }
    }
    else {
      print $nSMTP->message;
    }
    $nSMTP->quit;
  }
}
else {
  print "ERROR\n";
}
```

The only rule is that the CGI script must return "OK" as the response to the POST request so ns4 is able to see that the request succeeded. Any other result that the CGI script returns will be treated as a fail and will be used by ns4 as the reason of the failure.

Alerts are not automatically implied from the configuration for nodes and need to be specified under container options (like proxies) (as of v4.3.6 of ns4). Due to hierarchical inheritance rules, nodes will inherit container options which are defined at a higher level. This means that, by default, no alerts will be used unless they are defined on the root container under options with the “alert” attribute.

Note: ns4 has mandatory and optional Perl modules which are used depending on what the user has configured. Alerts fall within the category of optional Perl modules and to use the "alert_smtp" element you will need to ensure you have installed the LWP::UserAgent module.

6.2.6. Containers

Containers don't really contain any individual values but they enable groups to be formed to allow inheritance to occur. They are also mandatory as nodes must be defined within containers as container options refer to the nodes within the container.

For inheritance, containers can be nested within other containers - there isn't a limit on the amount of containers which can be nested. The only limit is what the user writing the configuration file deems as manageable. Below is a potential example of how containers can be used:

```
---
container:
 - # Parent Container
  container:
   - # PE Routers
    options:
     location: "Routers/PE"
   - # P Routers
    options:
     location: "Routers/P"
   - # CE Routers
    options:
     location: "Routers/CE"
  options:
   username: admin
   password: secret
```

In the above example all the nodes within the containers will inherit the *username* and *password* element, but they will define their own *location* elements.

6.2.7. Nodes

A node is an element which ns4 attempts to login to. This could be a router, switch or an appliance which supports a CLI. A node itself only has a very basic list of elements which must be defined under it (*id* and *address*) - the options for the nodes appear under container options which are where you define your cartridge, etc. Although they are called container options they are really node options as they are relevant to the node.

There isn't really much to the definition of a node as all the different bits which are associated with the node are defined elsewhere. There isn't any limitation on what nodes ns4 is able to deal with due to the ability to write cartridges. The only real limitation is they have to support a CLI.

id — This is the id/name of the node which can be accessed through the special variable [Node] so it is important that this name is found within the prompt of the node. When you write cartridges you generally use special variables or regex's within the prompt so it is important that the node has the correct prompt defined. This element will be treated as literal text so you don't need to escape any special regex characters. A simple validation is done on this value to ensure that it doesn't contain a "\", "/", "*", "?", ":" or any form of space.

address[:port] — This is the IP address of the node. However, this doesn't necessarily need to be an IP address and a hostname will also be accepted. You can also specify an optional port if you are connecting to a node connected to a terminal server line.

This value is optional and if omitted ns4 will use the "id" for the address. It will also output "DNS" as the address in report outputs.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element.

Taking the above node elements into consideration an example node configuration would look like:

```
---
container:
 node:
  -
   id: PE01
   address: 172.16.0.101
  -
   id: PE02
   address: 172.16.0.102
  -
   id: PE03
   address: 172.16.0.103
  -
   id: PE04
   address: 172.16.0.104
```

6.2.7.1. Dynamic Transformations

Within the configuration file we have the ability to dynamically create nodes without having to specify all of them statically if they have sequential numbering. To show how this works, let's assume we have a list of 10 routers which are aptly named R01, R02 upto R10 which have an IP address of 172.16.0.101, 172.16.0.102 upto 172.16.0.110. Normally we would have to define them individually which would take up time and space within the configuration file.

Dynamic transformations introduces the "{base, count[, step]}" syntax which can be used within the configuration file when there are nodes which follow a naming sequence. The "base" parameter is the starting value which is used on the first node with the "count" being the amount of nodes to dynamically create. The "step" parameter is optional and will default to 1 if omitted – negative values can also be used for the "step" parameter.

Instead of having to define 10 separate lines within the configuration file we could do the following:

```
---
container:
 node:
  - {id: "R{01,10}", address: "172.16.0.1{01,10}"}
```

In the above example we have used "01" as opposed to "1" – if a leading space is entered into the "base" parameter then this is maintained within the formatting when the numbers are outputted.

By using the above configuration it will now define 10 routers that all have the same properties and options that they would have had if they were all individually defined.

Note: When using dynamic transformations the "count" parameter must be consistent with all occurrences on a single line. If you specify 10 routers you can't specify a count of only 7 on the IP address otherwise there will be a mismatch. However, you could always omit the "address" and rely on DNS if the hostname/id exists within DNS.

Let's now assume we have R01 to R10 as previously, except now the IP addressing is reversed and R01 has an address of 172.16.0.110 and R10 having an IP address of 172.16.0.101:

```
---
container:
 node:
  - {id: "R{01,10}", address: "172.16.0.1{10,10,-1}"}
```

In the above example we have used the optional "step" parameter within the IP address to deduct one from every new node which will create our 10 routers, but will reversed IP addresses.

Note: Dynamic transformations are taken at face value which means if you use a "base" parameter of 254 with a "count" parameter of 25 then it won't automatically increment the third octet of the IP address when you reach 256. There is no limit on the amount of braces which can occur within a single line.

In summary, dynamic transformations can have potential value as a time and space saver, but it is very much dependant on the users naming and addressing scheme as they have to be sequential to take full advantage of this feature.

6.2.8. Container Options

Container options are used to specify different options associated with the nodes defined within the container. Although they are called container options they should really be called node options as they only contain options which are specific to nodes. The power of ns4 is in the ability to inherit container options on a per element basis which allows you to construct configuration files which gives you the most flexibility.

There are two types of container options: standard and custom. The standard options are defined within the core of ns4 and cannot be changed, but the custom options allow you to create user-defined options which you can then reference from within scripts and cartridges using the "[C:*]" syntax.

6.2.8.1. Standard Options

The following options are defined as of the latest version of ns4. They cannot be changed outside the main code and are deemed to be adequate for most nodes.

cartridge This is required to specify the id of the cartridge that is relevant for the nodes within this container.

username This specifies the username for the nodes within this container.

password This specifies the password for the nodes within this container.

method This is a string which determines what command is going to be run to connect to the node. It will consist of the binary (i.e. telnet/ssh) and the parameters to pass to it. We allow three different dynamic variables within brackets which can be used to construct this string: [Host], [Port], [User] (case insensitive).

This string is passed directly to the Unix "exec" function call, so you need to ensure that you place a comma between parameters. I am hoping that the following examples will demonstrate this:

"/usr/bin/telnet [host], [port]"
This will connect by running "/usr/bin/telnet" and it will substitute the [host] and [port] parameters for the real values at runtime.

"/usr/bin/ssh -p [port], [user]@[host]"
This will connect by running "/usr/bin/ssh" and it will substitute the [host], [port] and [user] parameters for the real values at runtime.

"/usr/bin/ssh -v, -p [port], -l[user], [host]"
This is an extension to the example above and demonstrates the flexibility in being able to enable ssh to be more vocal for you to diagnose any connection related issues. Each parameter needs to be separated by a comma.

Note: So ns4 is able to work out default port numbers the binary at the left of the string must contain either "ssh" or "telnet".

[transport] This allows you to specify the id of the transport that is relevant for the nodes within this container. You may specify more than one transport within this attribute to have the outputs sent to multiple transports.

```
---
container:
 options:
  transport: sftp_server
```

The above example demonstrates how to specify a transport for the nodes within the container. They will use the transport "sftp_server" which has been defined globally.

[alert] This allows you to specify the id of the alert that is relevant for the nodes within this container. You may specify more than one alert within this attribute to have the failed nodes list sent to multiple alerts.

```
---
container:
 options:
  alert: [pri_mail, sec_mail]
```

```
```

The above example demonstrates how to specify an alert for the nodes within the container. We have specified multiple alerts which will both be used by the nodes within this container.

[proxy]

This allows you to specify the id of the proxy that is relevant for the nodes within this container or allows proxy chaining by specifying multiple proxies. If you don't specify this option then ns4 will assume that you are using the null proxy and the nodes are directly accessible.

If we are going through a single proxy then we could define our cartridge options as follows:

```
---
container:
 options:
  proxy: ssh-proxy
```

However, if we wanted to daisy chain through multiple proxies then we could define it as follows:

```
---
container:
 options:
  proxy:
   -
    ssh-proxy
   -
    ts-proxy
```

Note: The connection will flow through the proxies in the order that they are specified.

When we define multiple proxies there are a few elements within the proxy which no longer have any meaning and are ignored. These only have a meaning if they are defined within the first proxy in the list:

rawpty
max_sessions
min_nodes_per_session
max_proxy_retries

The values within the first proxy within the list are used to define the characteristics of the connection.

We also have the ability to load balance across multiple proxies. This can be achieved using the "|" character to split up the proxies. This can be used if we have two different proxies which allow you to access the same nodes:

```
---
container:
 options:
  proxy: proxy-alpha|proxy-beta
```

In the above example we are sending 50% of the nodes through "proxy-alpha" and 50%

of the nodes through "proxy-beta". If we wanted to send 66% through "proxy-alpha" then we could specify "proxy-alpha" twice to weight it. To see what nodes are going through which proxy you can run ns4 with the "-r" parameter which outputs the report.

location

When you send output to a transport you can specify a location to enable ns4 to create a directory structure for your configurations. By default ns4 will place the configuration into a directory with the current date using the format "YYYYMMDD". By using this option you could extend that directory to include a name (i.e. YYYYMMDD/Routers/PE) if you used "Routers/PE" for a location.

[command]

This defines the list of commands which ns4 is going to run on the nodes which are defined within this container. This option allows you to specify a list of commands, but you need to ensure that you define an transport if you have defined commands.

The *command* element also supports the use of the *alias* element, which allows you to redefine the directory that is created on the transport server if you wish. You would use this if you have a complicated command that you wanted to shorten:

```
---
container:
 options:
  command:
   -
    show running-config
   -
    value: show ip cef hardware register-asic spiderpig location 0x01
    alias: show ip cef
```

In the above example instead of creating a really long directory to contain the output, it would get shortened to "show ip cef" which is much more manageable.

We also have the ability to enter sub-modes for running commands which need to be run from sub-modes. See the list option within the command line options for more details, but an example is included below:

```
---
container:
 options:
  command:
   -
    show running-config
   -
    "{admin, ^.*[Node]\(admin\)#}"
   -
    value: show running-config
    alias: admin show running-config
   -
    "{exit}"
```

Note: If we omit the prompt then it will use the default node prompt.

Note: The prompt within the above syntax for sub-modes is treated as a regular expression so you need to ensure that it has been escaped if you are using characters which could potentially be treated as regular expressions that you wouldn't want to.

Note: If you do enter a sub-mode then you need to ensure that you leave the sub-mode before ns4 attempts to finish otherwise it will not see the correct prompt that it expects.

[tag]	This allows you to define a list of tags which are associated with the node. You can then use these tags when you select nodes that you wish to execute commands on.
[gpg_keyid]	This allows you to encrypt the data using GnuPG before it is sent to the ftp servers. This option allows you to specify a list of gpg key ids which are defined within the local user's key ring. If you use this option you also need to ensure you define the global option "gpg_path".
max_node_retries (default: 3)	This defines the maximum amount of times that ns4 will attempt to login to a node before it declares the node as dead. One exception is when you are running a script on a node and you come across a script syntax error and then it won't attempt to retry.
timeout (default: 10 sec)	This is the timeout that ns4 will wait for an expected prompt before bombing out. The "max_proxy_retries" * "timeout" is used to determine how long ns4 will wait before determining a node is dead.
snmp_community	If you are using ns4 to poll SNMPs OIDs then you need to use this option to specify the community string that will be used to access the node.
[snmp_oids]	This value is used to specify the SNMP OIDs that are going to be polled. You need to define both the "id" and "oid" attribute where the "id" attribute is used to give it a name which is printed to the screen and the "oid" attribute is used to specify the actual OID. For an example, please see the documentation around the "-p" command line option.

Note: The **red** elements are mandatory elements and must be specified within every instance of the element. The brackets [] indicate that this element supports a list to specify multiple entries.

Taking the above container option elements into consideration an example container options configuration would look like:

```
---
container:
 options:
  cartridge: cisco-ios
  proxy: ssh-proxy
  username: admin
  password: secret
  method: "ssh [user]@[host]"
  command:
   -
    show running-config
   -
    show startup-config
  tag:
   -
    CISCO
   -
    PE
  snmp_oids:
   - {id: "Free Mem", oid: "1.3.6.1.4.1.9.9.48.1.1.1.6.1"}
   - {id: "CPU", oid: "1.3.6.1.4.1.9.9.109.1.1.1.1.7.1"}
  snmp_community: "CISCO"
```

6.2.8.2. Custom Options

By default there aren't any custom options as it is down to the user to define. Depending on the node which you are using depends on the type of custom options that you would decide to define. An example custom option which we

have used throughout our examples is the Cisco enable password. You would define that as a custom option under the container options and then you could implement it within cartridges using the "[C:Enable]" syntax.

You can define as many custom variables as you want and you can call them whatever you want. They are defined within a separate structure so they won't clash with any other variables within the schema. Custom options allow you to use the *encoding* element to encode the values as we don't know what you could be using custom options for. As an example you would define a custom option as follows:

```
---
container:
 options:
  custom:
   enable:
    encoding: rot13
    password: frperg
```

If we put all the above together from all the different sections then we can construct a basic configuration file which illustrates some of the different elements we have used throughout the examples within this document:

```
---
container:
 node:
  -
   id: PE01
   address: 172.16.0.101
  -
   id: PE02
   address: 172.16.0.102
  -
   id: PE03
   address: 172.16.0.103
  -
   id: PE04
   address: 172.16.0.104
 options:
  cartridge: cisco-ios
  location: "Routers/PE"
  proxy: ssh-proxy
  username: admin
  password:
   encoding: rot13
   value: frperg
  method: "ssh [user]@[host]"
  command: show running-config
  tag: CISCO
  custom:
   enable:
    encoding: rot13
    value: frperg

proxy:
 id: ssh-proxy
 address: 20.30.10.1
 method: "ssh [user]@[host]"
 username: root
 password:
  encoding: rot13
  value: frperg
 prompt: "^ssh-proxy:/"

transport_ftp:
 id: localhost
 address: 127.0.0.1
 username: guest
 password:
  encoding: rot13
  value: frperg
```

```
alert_smtp:
 id: smtp-relay
 relay: 20.30.40.56
 from: root@ns4-host-box
 to: alerts@my-company.org
```

7. Socks Wrapper

This is a new feature in ns4 which allows you to use SSH dynamic port forwarding to connect through proxies. Normally, ns4 will log into a proxy using telnet/ssh and then spawn telnet/ssh from the proxy to connect to the node. There is a one-to-one mapping between proxy and node which means you can only process a single node through a proxy at any given time. We are limited by the amount of connections that we can have through a proxy as networks are not designed to have lots of connections to the same place. For this reason we limit the amount of simultaneous connections to a proxy to 5 by default (this can be changed) which means we can only process 5 nodes at a time.

Secure Shell (SSH) has the ability to forward ports and this can be achieved by two mechanisms. Firstly, we can manually create tunnels on-demand or alternatively we can put SSH into dynamic mode (-D) which makes it act like a Socks proxy. ns4 uses the latter and will connect to a proxy and act as a Socks proxy which means we can connect to nodes through the Socks proxy. As we only have a single connection to the proxy it means we can increase the amount of simultaneous connections to be able to process more devices at the same time (by default this is 12).

To use this feature you need an additional program which is called a Socks Wrapper as SSH is not able to connect to nodes via a Socks proxy (even though it can act as a Socks proxy). Within the ns4 distribution in the "contrib/" directory is a file called "connect.c" which needs to be compiled on your system. This should be a relatively simple task as long as you have a compiler installed. Once you have compiled "connect.c" you need to place it into an accessible location (e.g. "/usr/local/bin").

Now that everything is in place for this feature to work, it is a good idea to give an overview of how this feature actually works. When ns4 identifies that a proxy has been configured to use the Socks Wrapper it identifies what nodes support this based on the following criteria:

1. Nodes' method is SSH
2. Proxies' method is SSH
3. Proxy supports port forwarding
4. We aren't using a multi-hop proxy
5. It isn't a null proxy

Once the above criteria has been satisfied ns4 connects to the proxy using the syntax specified within the "method" element, except it adds the "-D" parameter. By default, ns4 will use a random port for "-D", but you can override this by using the "local_socks_port" element. Once we have connected to the proxy and we are ready to accept Socks connections we spawn all children to connect to the nodes. Each node is connected to using the "method" element with the following parameters added: "-o ProxyCommand /usr/local/bin/connect –S 127.0.0.1:<PORT> <HOST> <PORT>". This makes SSH use the Socks Wrapper to connect to the node via the Socks proxy.

Two new elements have been added to the configuration file to support this feature: "socks_wrapper" and "local_socks_port". You can either enable the Socks Wrapper globally, or you can enable it on a per-proxy basis. It is advisable to enable it on a per-proxy basis and this won't work with some SSH proxies (this feature is still experimental, although has been tested using certain proxies). The "local_socks_port" option can only be specified under the proxy as this option is only relevant to the proxy and has to be different for every proxy.

In summary, all you need to do to enable the new Socks Wrapper mode is defined the "socks_wrapper" element under the proxy and ns4 will attempt to use this method for all nodes which match the criteria listed above. If you have a mix of nodes which support telnet and SSH going through the same proxy then ns4 will split them up and use the regular method for nodes using telnet and the Socks Wrapper mode for nodes using SSH.

8. Error Messages

This section details some of ns4's most common error messages that will be seen when connecting to nodes and tries to explain what they actually mean, although most of them I am hoping are pretty self explanatory:

8.1. Proxy Errors

Login Timed Out
This indicates that the login process timed out - this was during entering the username or password for the proxy server.

Invalid Username/Password
This indicates that ns4 saw either the "username" or "password" prompt twice which means the password or username was invalid.

Host Key Failed
This indicates that the local SSH host key that is stored on the local machine doesn't match what the proxy server is presenting. You will need to clean out the invalid entry within your "known_hosts" file.

Couldn't Spawn Command
This indicates that the command you have specified within the "method" under the proxy couldn't be spawned.

Port Forwarding Failed
This indicates that you have requested the proxy to use the Socks Wrapper, but when it attempted to act as a socks proxy using a dynamic port (either random, or locally specified) it failed to listen on that port.

8.2. Node Errors

Login Timed Out
This indicates that the login process timed out - this was during entering the username or password for the node.

Invalid Username/Password
This indicates that ns4 saw either the "username" or "password" prompt twice which means the password or username was invalid.

Buffer Space Exceeded
This indicates that ns4 got back more from a single command than was defined within "max_buffer_kb".

Host Key Failed
This indicates that the local SSH host key or SSH key stored on the proxy (if applicable) doesn't match what the node is presenting. You will need to clean out the invalid entry within your or the proxies "known_hosts" file.

Couldn't Spawn Command
This indicates that the command you have specified within the "method" under the node couldn't be spawned.

Cartridge Processing Prematurely Terminated
This indicates that the server responded with an EOF half way through processing the cartridge.

Script Not Suitable
This indicates that the script is not suitable for the node which has been selected. This should be defined within the $script_criteria variable at the top of your script.

Script Timed Out
This indicates that the script took longer than the interval defined within the $script_timeout variable at the beginning of your script. If you haven't defined it within your script then it will use the default of 120 seconds (2 minutes).

Pre/Post Command Failed

So we don't get into never ending loops, ns4 will only allow each element within the pre or post sections to be executed once. This error will be seen if an entry in the pre or post section gets matched twice.

Script Syntax Error

This indicates that a syntax error has been discovered within your script – this also indicates that your script most probably prematurely exited so some commands may have been executed on the node.

9. Shell Return Codes

When a Unix program exits then it will return a code which identifies if it was successful or not. From a Unix shell this value can be obtained by looking at the "$?" variable. Most programs don't bother setting the correct return code so you will always get back 0 regardless of what the program did. ns4 actually sets the return code so you are able to determine if ns4 was successful or not.

At the present moment in time ns4 supports the following shell return codes:

0 OK

1 Not OK

2 All Nodes Failed

5 Invalid Usage

10 Some Nodes Failed

10. Scripts

The true power of ns4 lies within its ability to use scripts to automate tasks within your network. Scripts are written in 100% pure Perl and are extended to support ns4 through the internal ScriptObject package which gives your scripts access to the context of the node. A script is a snippet of Perl code which is executed after ns4 logs into a node - you don't have to deal with the complex task of logging on or off from a node as that is handled by ns4.

Below is the standard template of a basic script which we will use as a reference while we explain how scripts work:

```
$script_version = <<SV;
My Router/Switch Script v1.0
Copyright (c) 2009 Chris Mason <chris\@noodles.org.uk>
SV

$ns4_version = "4.3.6";
$script_timeout = 120;
$script_criteria = ":";

my $global_var = "This is a script variable accessible within the script";

sub pre {
  print "This happens before we connect to any nodes\n";
  return (100);
}

sub main {
  print "This happens on every node\n";

  my $sO = new ScriptObject;
  $sO->cmd ("write memory");

  return (100);
}

sub post {
  print "This happens after we have connected to all nodes\n";
  return (100);
}

1;
```

Note: Although it isn't enforced the general recommended extension for an ns4 script is ".nss".

Note: All scripts must have a positive return code as the last line of the script. Scripts are handled like Perl modules so need to be able to see this to know that the script has been loaded successfully. By placing a "1;" as the last line in your script you will fulfil this criteria.

10.1. Pre-Defined Script Variables

There are currently four pre-defined script variables: $script_version, $ns4_version, $script_timeout and $script_criteria. None of these pre-defined script variables are mandatory - there are defaults for $script_timeout and $script_criteria of "120" and ":" respectively.

10.1.1. script_version

This will allow you to output a short version message before you script starts, either with the version of your script or a short message about your script.

```
$script_version = <<SV;
My Router/Switch Script v1.0
Copyright (c) 2009 Chris Mason <chris\@noodles.org.uk>
SV
```

When you execute the script on a node you will get the following output:

```
ns4 v4.3.6
Configuration Management Tool
url http://www.noodles.org.uk/ns4.html
by Chris Mason <chris@noodles.org.uk>

* My Router/Switch Script v1.0
* Copyright (c) 2009 Chris Mason <chris@noodles.org.uk>

...
```

10.1.2. script_criteria

This defines what nodes this script may run on. It allows you to stop someone from running a script designed for a Cisco router to be run on a Juniper firewall. It is very rare that a script will be compatible with two difference pieces of kit made by different vendors as their CLI will vary.

The syntax for the $script_criteria is as follows:

"X:Y[,X:Y,...]"

There are two parts to the above syntax with a mandatory colon separating them in the middle. The X indicates a regular expression which is used to specify nodes based upon a tag and the Y indicates a regular expression which is used to specify nodes based upon the node name. If you omit either the X or the Y then they are replaced with a match all regular expression.

By using a comma you can add multiple X:Y expressions that allow you to select multiple groups as the comma acts like an OR operator.

":"	This script can be run on all nodes as X (tag) will be evaluated as '.*' and Y (name) will be evaluated as '.+', which means all nodes would match. This is the default if you omit the $script_criteria definition.
"^PE\$:"	This script will only be allowed to run on nodes which are associated with the "PE" tag. Any node which has the "PE" tag will match as Y (name) will be evaluated as '.+'.
":^PP"	This script will only be allowed to run on nodes which start with the letters "PP". The tag (X) is irrelevant and will be treated as '.*'.
":^PE,:^PP"	This script will only be allowed to run on nodes which start with "PE" or "PP". However, this could have been written as ":^(PE|PP)".
"^PE\$:01\$,^PP\$:02\$"	This could have not been achieved without the use of the comma as we are selecting different nodes with different tags. This will allow all nodes with the tag of "PE" and ending in "01" to run this script as well as all nodes with the tag of "PP" and ending in "02".

Note: Due to the way scripts are loaded you need to escape the $ sign otherwise it is interpreted as a scalar in Perl. For example, "^PE$:" would have to be written as "^PE\$:".

10.1.3. ns4_version

This variable allows you to specify a minimum version of ns4 required to run this script. If this variable is omitted then there are no version restrictions. This can be used if you are writing scripts which utilise features which are found in later versions of ns4. The format of this variable should be in the format of the version number of ns4, i.e. "4.3.6".

10.1.4. script_timeout

This option allows you to set a limit (in seconds) on the amount of time a script will run on a node. There have been cases where a script could get stuck in a loop due to it being poorly written which would cause ns4 to freeze waiting indefinitely. We have now introduced an option which allows us to set a limit.

10.2. Script Defined Variables

Within scripts we also have the ability to define variables which are global to the scope of the script. These are defined as follows:

```
my $global_var = "fred";
```

Due to the architecture of ns4 there are a couple of restrictions regarding changing global variables. As the "pre" routine is run before we connect to any nodes we have the ability to modify these global variables and the changes will be seen within the "main" and "post" blocks. However, ns4 adopts a threaded model (forking) which means any changes made to these variables within the "main" block are not visible outside this block (i.e. in the "post" block).

There is also a caveat to this which should be taken into consideration – there are situations when multiple nodes are executed within the same forked process (e.g. if processing more than one node) – in this scenario future nodes within the same forked process will be able to see changes made to global variables as they are still within scope. To ensure that no unexpected results happen it is recommended **not** to change the value of a global variable within the "main" block.

10.3. sub pre()

The "pre" subroutine is optional unlike the "main" subroutine and allows you to run portions of code before you connect to any nodes. An example of where this could be required would be if you wanted to create a configuration repository, where the "pre" subroutine could be used to check for its existence and if it didn't exist then it could create it.

10.4. sub main()

The "main" subroutine is where the heart of the script is as it contains the script which is actually executed on each node. This would be where you would run commands and fetch output from nodes. Unlike the "pre" and "post" subroutines the "main" subroutine is run once on every node once you have logged on.

10.5. sub post()

The "post" subroutine is optional unlike the "main" subroutine and is like the "pre" routine, except it allows you to run portions of code after you have processed the nodes. An example of where this could be required would be if you wanted to email a report based on the output of the nodes. Each node could write data to a temporary file within the "main" subroutine and then you could check to see if that file exists and then email it in the "post" routine.

10.6. Return Codes

The "pre", "main" and "post" subroutines must return a value at the end of the subroutine for the script to succeed. There are a few options which can be used here.

1. 'return (100);' – this will ensure that an "OK" status code is passed back, however it lacks the ability to pass a message.

2. 'return ("100:Stats Updated");' – this allows us to pass through an "OK" status code, but ns4 will display the message contained after the colon within the return statement.

3. 'return ("Something Went Wrong");' – the final option is to return a status message which doesn't begin with "100" which will result in a Custom Script Response (CSR) being presented which indicates an error occurred.

If the "pre" block doesn't respond with an "OK" status code then the "main" and "post" blocks are skipped.

10.7. ScriptObject

A ScriptObject is a package within ns4 which you can create an instance of, from within your script. From that instance you are able to access the various methods of the ScriptObject to control the node. Without the ScriptObject instance you have no way to talk to the program and your script will just be a simple Perl script with no ns4 interactions.

Note: The ScriptObject can be instantiated anywhere within the script – even created once globally, however certain methods can only be executed from within the "main" block. These are methods which require connectivity to a node to succeed, i.e. "cmd" method or "dvar" method with a node specific dynamic variable.

10.7.1. new

```
my $sO = new ScriptObject;
```

Creates a new ScriptObject instance. This command doesn't take any parameters and returns a new instance of the ScriptObject package.

10.7.2. cmd

```
my @sR = $sO->cmd ("show running-config", [Prompt => ...], [Timeout => ...]);

or

my $sR = $sO->cmd ("show running-config", [Prompt => ...], [Timeout => ...]);
```

After you have created a ScriptObject instance you can use the "cmd" method to run a command on a node. The return value of the "cmd" method is context sensitive depending on the lvalue of the function call. If you specify an array then ns4 will split each line up within the output as opposed to a scalar where ns4 will join the content with "CRLF".

There are two optional parameters which can be passed to the "cmd" method: the "Prompt" (case sensitive) parameter allows you to specify an alternative prompt if it is not going to be the default prompt and the "Timeout" (case sensitive) parameter which allows you to redefine the timeout for that command.

Generally, the default node timeout is adequate for most commands and can be omitted - the only exception is if you are trying to fetch a large output which takes a long time before it starts sending output. When you receive content from the node the timeout is reset, so it is a time in seconds that passes where content has not been received. The prompt can also be omitted if it matches what is defined within the cartridge as the normal prompt that you logged in with.

$sO->cmd ("write");

This will execute the command "write" on a Cisco router. This command doesn't prompt you for anything and it will save the configuration into NVRAM so it isn't lost if the router was reloaded. We haven't assigned the output to a variable as we don't care what the output is and we haven't specified a prompt as it doesn't change when we run this command.

my @r = $sO->cmd ("show clns neighbor | include Up");

In the above example we are looking for CLNS neighbours which are currently in the "Up" state. We are also assigning the output to the variable "@r" so we are able to parse it using Perl. We have returned it as an array so we don't have to split the contents. As an example below I have included the output from this command first, then shown how we could parse it in Perl:

```
PE01# show clns neighbors | include Up
PP02       Et0/0          aabb.cc00.6600      Up     26   L1   IS-IS
PP07       Et3/0          aabb.cc00.6b03      Up     28   L1   IS-IS
```

```
...
my @r = $sO->cmd ("show clns neighbors | include Up");

foreach my $l (@r) {
  if ($l =~ m/^(\S+)/) {
    print $sO->dvar ("Node") . " has a neighbor of $1\n";
  }
}
...
```

We can also see in the above example the use of special variables to output context sensitive information.

$sO->cmd ("configure terminal", Prompt => "^.*\Q" . $sO->dvar ("Node") . "(config)#\E");

We have specified a prompt this time because when we enter configuration mode the prompt changes on a Cisco router. We are using special variables to dynamically place the node name into the prompt and we have had to escape the meaning of the brackets otherwise they would be interpreted as a special character.

This allows us to create dynamic configuration scripts which we can run on multiple nodes. Below is an example which could be used on a Cisco router (we have used the MATCH_X special variable to make the syntax simpler):

```
...
my $nPrompt = $sO->dvar ("MATCH_L*,N:Prompt,#");
$sO->cmd ("configure terminal", Prompt => $nPrompt . "\Q(config)\E#");
$sO->cmd ("line vty 0 4", Prompt => $nPrompt . "\Q(config-line)\E#");
$sO->cmd ("exec-timeout 30 0", Prompt => $nPrompt . "\Q(config-line)\E#");
$sO->cmd ("end");
...
```

We don't need to specify a prompt when we use the "end" command as it takes us out of configuration mode and back to the default prompt we saw when we logged in.

However, there is an alternative if you don't fancy having to redefine the prompt every time that you change sub-modes. You can specify a wildcard within the actual prompt within the cartridge so it doesn't matter what sub-mode you enter as the regular expression still matches:

^.*[Node].*#

The above syntax will allow any character to appear after the prompt name. When using the above syntax you could omit the Prompt parameter from the "cmd" method. However, you then assume that every command is going to be executed successfully and you are always going to enter the new sub-mode.

my @r = $sO->cmd ("show running-config", Timeout => 60);

As we have seen this command take a while on a busy Cisco router we have decided to change the default timeout to 60 seconds. However, we don't want to specify a prompt as we know it doesn't change when we use this command. We have also assigned the output to "@r" so we can parse it later.

Note: If any of the commands timeout then ns4 will handle that and break out of the script immediately.

Note: The "Prompt" parameter expects a regular expression so ensure that if you are passing in the node name that you escape any special characters. For example "PE01-CISCO" would have to be passed in as "PE01\-CISCO" which you can achieve using the "\Q" and "\E" meta characters.

10.7.3. svar

```
print "Script Variables 'mtu' is '" . $sO->svar ("mtu") . "'\n";
```

The "svar" method allows you to access script variables which have been passed through on the command line. In the above example we have used the variable "mtu" which would have been passed through using the following syntax:

```
$ ns4 -n PE01 -s myscript.nss,mtu=1500
```

Note: If the script variable doesn't exist then the "svar" method will return "undef".

10.7.4. dvar

```
my $sR = $sO->cmd ("show running-config", Prompt => $sO->dvar ("N:Prompt"));
```

The "dvar" method allows you to access special variables like you would normally do using square brackets within cartridges. Instead of using brackets you would use the "dvar" method. If the special variable doesn't exist then the method will return "undef". The "dvar" method also supports the more complex MATCH_X special variables which could be used in the following context:

```
$sO->cmd ("configure terminal", Prompt => $sO->dvar ("MATCH_L*,N:Prompt,#") . "\Q(config)#\E");
```

The above example also shows the concept of escaping characters using the "\Q" and "\E" meta characters as it is being treated as a regular expression. The "N:Prompt" special variable doesn't want to be escaped as it should be treated as a regular expression. If you didn't use the "N:Prompt" special variable and constructed the prompt manually you would have to escape the special variables:

```
$sO->cmd ("configure terminal", Prompt => "^.*\Q" . $sO->dvar ("Node") . "#(config)\E");
```

The "dvar" method is also used for custom special variables which are defined within container options and can be used as follows:

```
print "Node is " . $sO->dvar ("Node") . "\n";
print "Node Type is " . $sO->dvar ("C:Type") . "\n";
```

10.7.5. break

```
return ("Break Detected") if ($sO->break);
```

The "break" method allows us to check if CTRL-C has been pressed by the user. ns4 will always finish scripts when a break sequence has been detected, but if you want the ability to terminate your script early you can check for the break sequence using the "break" method.

10.7.6. lock

```
if ($sO->lock ([timeout])) {
  # Do something exclusively
  $sO->unlock;
}
else {
  return ("Lock Failed");
}
```

As the same script can be running on multiple nodes in parallel due to the threaded architecture of ns4 it is important to understand the implications this can have on output and writing to files. If multiple threads attempt to write to the same file at the same time then you will end up with muddled up text and in the worst case scenario – missing text.

The "lock" subroutine provides a function to obtain a mutex lock where you can get exclusive access while you have obtained the lock. This stops other parallel processes from doing the same thing at the same time. Once you have finished you "unlock" which releases the mutex lock for another process to obtain it.

Warning: It is best practice to only do the absolutely minimum that is necessary within the mutex lock otherwise other threads/processes are starved and will halt while they wait for the mutex lock. An optional "timeout" value can be specified which is the time it will wait for the lock until it fails. If this parameter isn't specified then a default value of 5 seconds is used.

Warning: Check all the execution paths of your code and ensure that you use the "unlock" subroutine otherwise you will cause lock timeouts. If you return from within the lock you need to ensure you "unlock" first!

10.7.7. unlock

See "lock".

10.7.8. create_config_tree

```
my @rc = $sO->cmd ("show running-config");
my $ct = $sO->create_config_tree (\@rc);

or

my $ct = $sO->create_config_tree ("config.txt");
```

The above function is used to create a hash tree of a configuration file. It analyses a configuration file and based on indentation will create a tree hierarchy. It is mostly used by the "cdiff.nss" script, but I have added it to ns4 as it contains some nice functions which could be reused in other scripts.

This function together with "output_config_tree" can be used to normalize the formatting of configurations of routers. It can be used to sanitise the amount of spaces within indentations as well as removing comments and whitespace.

The subroutine returns a reference to a hash of hashes which either can be outputted using the aforementioned "output_config_tree" subroutine or could be used in the "diff_configs" subroutine.

10.7.9. output_config_tree

```
$sO->output_config_tree ($ct);

or

$sO->output_config_tree ($ct, $fh);
```

This function takes a reference to a hash tree as generated by "create_config_tree" and will either output it to the screen or to a file. If outputted to a file then you pass through a reference to an open file handle. If multiple routers are being processed then please see the "lock" and "unlock" subroutines for creating a mutex lock.

10.7.10. diff_configs

```
my $diff = $sO->diff_configs ($ct_a, $ct_b);
```

This subroutine is used to create a functional hierarchical diff between two configuration hash trees which have been generated by "create_config_tree". It will compute which elements have been removed or added and will create an output diff which will contain all the hierarchy to easily identify which sections a configuration line has come from. This subroutine will return a reference to a config diff hash which can be outputted using the "output_config_diff" subroutine.

10.7.11. output_config_diff

```
$sO->output_config_diff ($diff);

or

$sO->output_config_diff ($diff, $fh);
```

This subroutine is identical to "output_config_tree" except it outputs a configuration diff as opposed to a tree. If outputted to a file then you pass through a reference to an open file handle. If multiple routers are being processed then please see the "lock" and "unlock" subroutines for creating a mutex lock.

10.8. Demonstration Script

When we take all the above into consideration we can come up with the following example to demonstrate a script utilising all the different types of syntax:

```
$script_version = <<SV;
My Demonstration Script v1.0
Copyright (c) 2009 Chris Mason <chris\@noodles.org.uk>
SV

my $sMTU = 1500;
my $sO = new ScriptObject;

sub pre {
  if (defined ($sO->svar ("mtu"))) {
    $sMTU = $sO->svar ("mtu");
  }
  else {
    print "Warning: Script Variable 'mtu' Not Defined - Using Default\n";
```

```
  }
  100;
}

sub main {
  my $nPrompt = $sO->dvar (“MATCH_L*,n:prompt,#");
  my $nID = $sO->dvar (“node”);

  my @r = $sO->cmd ("show clns neighbors | include IS-IS");

  $sO->cmd ("configure terminal", Prompt => $nPrompt . "\Q(config)\E#");

  foreach my $l (@r) {
    return ("Break Detected") if ($sO->break);
    if ($l =~ m/^(\S+)\s+(\S+)/) {
      $sO->cmd ("interface $2", Prompt => $nPrompt . "\Q(config-if)\E#");
      $sO->cmd ("description >>> $nID ($2) to $1 <<<", Prompt => $nPrompt . "\Q(config-if)\E#");
      $sO->cmd ("mtu " . $sMTU, Prompt => $nPrompt . "\Q(config-if)\E#");
      $sO->cmd ("exit", Prompt => $nPrompt . "\Q(config)\E#");
    }
  }

  $sO->cmd ("end");
  $sO->cmd ("write");
  100;
}

1;
```

The above script logs into a Cisco router and formats the descriptions on all the core facing IS-IS interfaces. It also changes the MTU to either 1500 or the value that is passed through in the "mtu" script option. It then finishes by saving the configuration to NVRAM.

Note: As stated above the value which you specify within the "Prompt" parameter is very much dependent on what the prompt is defined as within the cartridge. By using the MATCH_X special variable with the "N:Prompt" value you need to ensure that the prompt you construct is going to be compatible with what you have defined within the cartridge.