**DFM: DFM ANT**

**Prepared By Mohan Komarthi**

===========================================================================

DFM ANT is the DFM Team's Automated Nightly Test framework and suite of tests. This framework provides conveniences and standards that allow the DFM Team to write and run automated test cases.

The flow of this document is for a person who is new to ANT and is going to start writing ANT tests. However, the doc assumes the reader is familiar with DFM, Perforce and perl. It starts off explaining how to setup the ANT environment, then how to run a test, then how to write a test, and concludes by describing available debugging tools. There's also a FAQ section.

• ANT Setup

• ANT Setup on Linux

• ANT Setup on Windows

• Remote ANT (RANT) Setup

• Running Tests

• Generic ANT Run time Environment definition

• Using a Valid ANT Run time Environment

• More details on running ANT

• Typical Invocation

• Output From Running Tests

• Getting More Output

• Debugging

• The Environment File

• Filing Burts

• Writing Tests

• A Simple Test Case

• A More Complex Test Case

• Test Module Structure

• Writing ZAPI tests

• Best Practices for Writing ANT Test Cases

• The Resource File

• Adding To The Resource File

• Using Scratch Objects In Your Tests

If you are going to be writing tests then you may want to have a copy of the ANT reference pages handy.

ANT's design objectives were:

• ANT should provide a simple and easy-to-use framework that offers convenience and consistency for writing and running test cases.

• Tests should be easy to write and maintain.

• Tests should be easy to run.

• Test results should be 100% reproducible.

• Tests should not rely on live hardware being in some known state.

• Tests should be capable of testing not only the format of, but also the content of command output that contains variables, such as reports.

In addition to improving the tests themselves, we would like to improve the test results reporting mechanism, to make test results more accessible and more meaningful. Towards this end we have additional goals for test reporting:

• Test results should be browsable in a GUI form.

• Developers should be able to annotate test results.

• Test reporting should include history, including when was the last time a test result changed.

This document is written for releases newer than NightTrain. It may be accurate for earlier releases but we have not attempted to verify the accuracy.

3. ANT Setup

DFM ANT runs on the same system as the DFM Server you will be testing. Do your setup steps on the system where the DFM Server is, or will be, installed. There are three parts to setting up the ANT environment. Once you complete these three parts you will be able to run and write ANT tests. The three parts are

• Install and launch the DFM Server you want to test.

• Install perl on the same system.

• Create your DFM ANT perforce client.

DFM ANT code is in perforce, just like the DFM source code. As a matter of fact, the ANT code is in the test2/ directory at the top level of the DFM branch. Whether you are going to run test cases, write test cases or both, you'll need to setup your ANT environment. You should set up your ANT environment on the same system as the DFM Server you'll be testing.

3.1 ANT Setup on Linux

Install the DFM Server just as you would normally. Make sure that you have full privileges (your account has the DFM role GlobalFullControl).

DFM ANT relies on perl 5.8.8 or newer. All of engineering's linux systems have access to perl5.8.8 or newer so you should not have to worry about this if you are going to be using linux. You can verify perl is available by running perl -version on the test system.

Note: On Fedora GNU/Linux/VMware simulators, perl sometimes comes from /usr/local/bin/ and this has a problem with Net::SSLeay package. It has been found that using /usr/software/bin/perl or /usr/bin/perl works better. If you run into problems, try a different perl and see if that works better.

The way to get ANT is to get the complete DFM branch and follow the steps described in the build guild. If you follow the build guild steps you will find the ANT files in the <top\_level>/test2/ directory. Make sure the perl sync\_shared command is executed and you have files in the <top\_level>/test2/zapi/ directory after perl sync\_shared has run.

Verify the setup is correct. Make sure the user, biff in this case,

your login in your case,

has GlobalFullControl and the DFM Server is completely up and running.

If you don't get these results then double check your steps and if problems

remain then take a look at the FAQ section.

linuxbox> dfm user role list biff

Role Id Role Name

------- -------------------------

19 GlobalFullControl

linuxbox> dfm services list

sql: started

eventd: started

monitor: started

scheduler: started

server: started

watchdog: started

http: started

linuxbox> cd /u/biff/p4/isis/test2

linuxbox> perl dfmtest.pl run -m Version

=== Welcome to DFM ANT ===

Running on Linux

Host: linuxbox

User: biff

DFM version: 3.8

DFM dir: /opt/NTAPdfm

Running tests...

Version:List:VersionUNIX ............................................. PASS

Version:List:VersionList ............................................. PASS

Test Results:

PASS: 2

The ANT environment is set up correctly.

3.2 ANT Setup on Windows

Install the DFM Server just as you would normally.

ANT relies on perl so you'll need to make sure perl is installed on the Windows system where you'll run the DFM Server and ANT tests. Our nightly run time environment on Windows uses ActiveState perl 5.8.8.

3.2.1 Setting up perl on Windows

The preferred method is to use an existing Perl distribution mounted from a shared directory. Shown below are recommended shared directories for various locations.

RTP:

C:\> net use X: \rtpinf02\localtest\arch\win32 /u:rtpinf02\minwin qa4testing

C:\Users\Administrator> net use X:

Local name X:

Remote name \rtpinf02.rtp.netapp.com\localtest\arch\win32

Resource type Disk

Status OK

# Opens 0

# Connections 1

The command completed successfully.

C:\Users\Administrator> type C:\mountall.bat

net use X: \rtpinf02.rtp.netapp.com\localtest\arch\win32 /u:rtpinf02\minwin qa4testing /persistent:no

C:\Users\Administrator>

SVL:

C:\local\bin> net use X: \svlinf06\localtest\arch\win32 /u:svlinf06\minwin qa4testing

C:\local\bin> net use X:

Local name X:

Remote name \svlinf06.eng.netapp.com\localtest\arch\win32

Resource type Disk

The command completed successfully.

C:\local\bin>

You can either configure these shares as persistent, or use a batch file such as the one shown above to establish the mapping. By convention, the shared directory is mapped to drive X:. You will also need to add some directories to the PATH environment variable:

• X:\perl588\bin

• X:\bin

(The second item above is for various MinWin utilities.)

If perl is already installed on your system, make sure that the first path above precedes any path which references the locally-installed perl. Here is an example from a DFM testbed system running Windows Server 2008 R2:

C:\Users\Administrator> echo.& echo Path=%Path%

Path=C:\Program Files\NetApp\DataFabric Manager\DFM\java\bin;C:\Program Files\Ne

tApp\DataFabric Manager\DFM\bin;C:\Program Files\NetApp\DataFabric Manager\Sybas

e\ASA\win32;X:\perl588\bin;C:\Windows\system32;C:\Windows;C:\local\bin;C:\Window

s\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\SUA\commo

n\;C:\Windows\SUA\usr\lib;X:\bin;X:\cygwin\bin;X:\pstools;C:\Windows\idmu\common

C:\Users\Administrator>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DEPRECATED

If your Windows system does not have perl then you can get a copy from www.ActiveState.com and put it on your Windows system. The most recent 5.8 release should work fine.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DFM ANT now requires the Sendmail.pm package. If you receive an error that this package cannot be found, you are running a different version of perl than that recommended above. Please re-check your PATH setting and correct it as necessary. When everything is working correctly, you should be able to run the commands below with identical results.

C:\> for %p in (perl.exe) do @echo %~$PATH:p

X:\perl588\bin\perl.exe

C:\> perl -v

This is perl, v5.8.8 built for MSWin32-x86-multi-thread

(with 50 registered patches, see perl -V for more detail)

Copyright 1987-2006, Larry Wall

Binary build 820 [274739] provided by ActiveState http://www.ActiveState.com

Built Jan 23 2007 15:57:46

Perl may be copied only under the terms of either the Artistic License or the

GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on

this system using "man perl" or "perldoc perl". If you have access to the

Internet, point your browser at http://www.perl.org/, the Perl Home Page.

3.2.2 Setting up Perforce on Windows

There may be other ways to accomplish setting up the Perforce client on Windows but this way appears to work.

If necessary, install Perforce on the Windows system. I've not gone through this process as all the Windows systems I've used for ANT had Perforce already set up. If you need to do this, then I recommend reading the HitchHiker's Guide. You can start with Using the Windows Clients.

Here's how a user would set up a client on Windows. All steps are done in a command window, as <HOSTNAME>\Administrator. The directory name (shown here as <isis-win>) may be chosen at will; we recommend a name that matches the codeline (branch) which you will be testing.

C:\> mkdir isis-win

C:\> cd isis-win

C:\isis-win> p4 client biff:isis-win

Biff sets the View

View:

//depot/prod/champagne/champagne-isis/... //biff:isis-win/...

C:\isis-win> p4 set P4CLIENT=biff:isis-win

C:\isis-win> p4 sync

C:\isis-win> perl sync\_shared

Verify the setup is correct. On Windows, DFM Server must be installed by Administrator, and both DFM and ANT run as Administrator. (This is different from Linux, on which ANT tests are run by a fully privileged non-root user.) On Windows, ANT runs as Administrator.

Verify that your DFM server is up and running, then try executing a test. If you don't get results similar to these, double check your steps; if problems persist, then take a look at the FAQ section.

C:\> cd isis-win

C:\isis-win> dfm user role list Administrator

Role Id Role Name

------- -------------------------

19 GlobalFullControl

C:\isis-win> dfm services list

sql: started

eventd: started

monitor: started

scheduler: started

server: started

watchdog: started

http: started

C:\isis-win> cd test2

C:\isis-win\test2> perl dfmtest.pl run -m Version

=== Welcome to DFM ANT ===

Running on Windows

Host: sig

User: SIG\Administrator

DFM version: 3.8

DFM dir: c:/Program Files/Network Appliance/DataFabric/DFM

Running tests...

Version:List:VersionWindows\_dfm.exe .................................. PASS

Version:List:VersionWindows\_dfmcheck.exe ............................. PASS

Version:List:VersionWindows\_dfmconfig.exe ............................ PASS

Version:List:VersionWindows\_dfmmonitor.exe ........................... PASS

Version:List:VersionWindows\_dfmperf.exe .............................. PASS

Version:List:VersionWindows\_dfmwatchdog.exe .......................... PASS

Version:List:VersionWindows\_eventd.exe ............................... PASS

Version:List:VersionWindows\_grapher.exe .............................. PASS

Version:List:VersionList ............................................. PASS

Test Results:

PASS: 9

If you get similar results, congratulations - you've finished setting up your Windows DFM ANT test environment!

3.3 Remote ANT (RANT) Setup

Remote ANT is a more generic name for what we currently call DFM ANT Torture. The basic idea behind Torture is to set up multiple ANT runs, all hitting a specific DFM Server. The Torturer (DFM ANT Torture) is running on a solaris system. The torturee is some other system (thus Remote in the sense that it is remote from DFM ANT) where all the ANT tests are running. The torturer might fire off 10, 20, or more test in parallel against the remote system. More details on Torture and Remote ANT can be found here.

4. Running Tests

There are two parts to successfully running ANT tests. The first part is getting all the necessary files from Perforce. That is described in the previous section, ANT Setup. The other part is the run time environment for ANT tests. That's what we discuss next.

In order to make running ANT tests simple and portable, an ANT run time environment has been defined in generic terms. When you run ANT tests, you'll want to make sure you are running in an actual run time environment that meets the requirements of this generic environment. First, we will look at the generic environment definition. Then we will see how to ensure you have a specific instance of the generic environment.

4.1 Generic ANT Run time Environment definition

The generic environment is an abstraction of the objects and resources you will find in the actual resource files used for the official ANT runs (eg. rsrc.A.svl, rsrc.B.svl, rsrc.C.svl in the test2/ directory). If you look at one of these resource files while reading this section you will quickly get the idea of how they are used.

The generic environment is (for DFM 4.0):

• Three filers running 7.3.2 or newer 7.x Ontap.

• All three filers have the SnapMirror license.

• One filer has SnapVault Primary license, but not SnapVault Secondary license, and there are two qtrees that can be used as SV primaries. The two qtrees should have some data, at least a few megabytes.

• One filer has SnapVault Secondary license, but not SnapVault Primary license, and there are two volumes that can be used as SV secondary volumes. These volumes need to be large enough that the SV primary qtrees can be backed up to them.

• There are at least 3 luns on at least 2 of the filers.

• There is at least one scratch aggr on the two SnapVault licensed filers. (See Using Scratch Objects In Your Tests for an explanation of scratch objects).

• There is at least one scratch traditional volume on each of the two SnapVault licensed filers.

• There is at least four scratch flex volumes on each of the two SnapVault licensed filers. These scratch flex vols are not on the scratch aggr nor the scratch trad vol. Each volumes should be at least 1 gigabyte.

• There is at least 20 scratch qtrees, spread across 2 volumes, on each of the two SnapVault licensed filers. These 20+ scratch qtrees should not be on the scratch volumes. The qtrees should reside on their own volumes. I usually create three 1GB flex vols on a filer and then create 6, 6, and 8 qtrees across those 3 vols.

• One of the filers has an RLM interface and a DNS assigned IP address for it.

• Three vFilers running on filers running 7.3.2 or newer 7.x Ontap. One of the vFilers needs to be on a filer with the SnapVault Primary license and the vFiler should have at least 2 scratch flex vols and 4 scratch qtrees. The other 2 vFilers need at least one volume each.

• Three hosts (ideally, one Windows, one Linux, and one Solaris) with OSSV and NHA installed. Each of these three systems needs to have two paths that can be walked by SRM. Each of these three systems needs two or more OSSV dirs.

• A few tests in the Group.pm test module will fail if DFM is monitoring less than 7 filers so beyond the three listed in the resource file, either add 4 more by hand or have 4 more in the resource file.

• There are some new tests that expect a c-mode cluster so be sure DFM is monitoring at least 1 cluster.

• The host system where DFM is installed.

4.2 Using a Valid ANT Run time Environment

So that's the generic run time environment. Now if you look at one of the actual resource files, you'll see they contain details for real instances of the generic resources. We have resource files for Sunnyvale environments and we have resource files for the RTP environments. The Sunnyvale environments are for ANT test development and can be used by anyone. The RTP environments and the corresponding resource files are for the official ANT nightly runs. The RTP environments are not to be used by anyone. Only the Sunnyvale environments are available for your ANT test development and execution.

In Sunnyvale, we have configured a set of filers and hosts to correspond to the three rsrc.X.svl files. These ANT environments can be signed out through thehardware signout page, under the SVL-ANT-ENV selection, or you can use the convenient auto-signout process. The auto-signout was introduced in the December 1, 2009 DFM ANT News announcement and is explained in greater detail here.

Here's a simple example of using the auto-signout method from a linux system. Simply invoke /usr/local/test/bin/dfmant run <module\_list>. The format of<module\_list> is module:[group][,module:group]\*. In other words, the list can have one or more module names, separated by commas. If no group name is listed then the whole module will be executed. If a group name is given, then only that group of tests within the module will be executed. The same command is available on Windows systems where MinWin has been installed. For these systems the path to dfmant is X:\bin\dfmant.

Another option is to signout one of the SVL environments, and then use the the corresponding rsrc.X.svl resource file in your invocation of ANT. The steps are:

a. Go to the hardware signout page and reserve one of the ANT envs.

b. Invoke ANT with the -z option. This is key. Using the -z option, along with the -r resource\_file option, will tell ANT to verify the contents of the resource\_file exist and make sure DFM is aware of the resources. This gives your ANT run a much better chance of passing. Note that you only need to use -z option the first time you run ANT after a fresh install of DFM.

Here's an example, after reserving the ANT\_SVL\_ENV\_A\_48HR\_MAX

environment for 48 hours.

mgt-ln130> cd /u/marlon/p4/champ/test2

mgt-ln130> perl dfmtest.pl run -r rsrc.A.svl -z -m Aggr:Syntax,BMHelp,BMOption

.

.

.

The next section has more details on the -z option and when to use it.

Both of these methods have a hidden assumption. If you look back in the description of the generic run time environment, you'll see that part of the environment is the host system where DFM is installed. It is likely that your DFM host is not included in the resource file you used while running your tests. In the example above, my DFM host, mgt-ln130, is not in rsrc.A.svl. So how did the ANT run work ? It worked because mgt-ln130 uses the root password that DFM ANT assumes is the default for all hosts, namely sundance.

If mgt-ln130 used some other password for root, I would have two options. Either create my own resource file or use an existing resource file, but on the command line I would need to supply the root password for mgt-ln130. For the auto-signout method, you can use the -a <user>:<password> option. If you are invoking ANT using the perl dfmtest.pl ... cli then you can include the root password in this fasion perl dfmtest.pl run -r rsrc.A.svl:ROOTPASSWORD=<notsundance> -m Aggr.

4.3 More details on running ANT

Test cases are contained within a test module file. The standard location for test modules is the test2/modules/ directory. The standard naming convention of modules is <mod\_name>.pm. Inside a test module file is one or more test groups. The individual test cases are inside a test groups. ANT notation uses : as a separator between the module name, group name and test case name. So Version:List:VersionUNIX is the complete test case name and it consists of the Versionmodule, the List test group inside Version, and VersionUNIX is the actual test case.

Each test case exits with either a PASS, a FAIL, or a SKIP status. PASS and FAIL are obvious. SKIP means the test was skipped for some reason. Possible reasons include things like the test expected DFM to have at least one agent (dfm agent list) but DFM did not have any so the test case decides to exit with SKIP rather than PASS or FAIL. The test case decides to exit with SKIP status.

ANT provides a way to specify which test cases to run. Options include running all the tests in the modules/ directory, or some subset of those tests. Note that a test module does not have to already be in perforce in order to run it. This allows you to run tests that are still in the development phase.

perl dfmtest.pl run is the simplest invocation to run ANT tests. This invocation will run all ANT tests in the modules/ directory. As of June, 2009, this command on linux runs over 5,600 test cases and takes over 20 hours. You probably don't want to run this command. Below are some of the options that help refine what tests are executed. The following link provides the full list of dfmtest.pl options.

Option Description

-a <module\_args> A list of arguments sent to the modules. The default is no arguments.

-c <conf-file> The log4perl config file to use. The default is ant.logger.conf. See Output From Running Tests for more details on how to use the logging facilitiess.

-d <dir> Location of modules. The default is ./modules.

-e Creates a file containing the current test environment. More details are given here.

-g <pattern> A pattern to match for test groups to run. Test groups matching this pattern will be run. The default is to match everything.

-h <dfm\_server> Specifies the remote dfm server to run against. You must ensure that there is a valid entry for the server in the resource file specified with the -r option. A list of test modules capable of being run against a remote server can be found in test2/torture.template.

-l Sets Audit logging Automation Verification on. Default is off.

-m <list> A comma separated list of modules to run. Do not include the .pm extension. The default is to run all modules in the -d <dir> directory.

-r <rsrc\_file> Specifies the resource file to use while running the tests. Details on resource files are provided later on. There is no default resource file.

-s This option takes no arguments and creates a silent run of dfmtest.pl. No printing other than perl errors and warnings will take place, and no files will be created. This option is used in conjuction with torture runs.

-t <pattern> A pattern to match for test names to run. Test cases matching this pattern will be run. The default is to match everything.

-x Tracks the execution time (in seconds) of each test.

-z If a resource file is being used, then attempt to guarantee the resources listed in the file really do exist and dfm is aware of them. If you are going to be running ANT multiple times with the same DFM Server then you probably only need to use this option the first time you invoke ANT. Look here for more details.

-T <tag1,tag2,..> Run just the modules that contain all the tag(s) listed. We can insert one or more AntTag: <tag> into a test module and subsequently have all of them run in a single ANT run. For example, we could add a # AntTag: ProvMgr tag to all the modules that test the Provisioning Manager. Then if we wanted to run just these tests we would use -T ProvMgr and ANT would execute the modules with that tag. Look herefor more details.

These two files are used to absolutely deny tests from running. If a test matches any of the patterns in either of these two files then the test will absolutely not be executed.

disabled\_patterns.txt This is not an option, but it influences which tests get executed. Any module name or test case that matches a pattern in this file will not be run.

disabled\_tests.txt This is not an option, but it influences which tests get executed. Give the full test case name, <mod>:<grp>:<test> of the test case not to run.

4.4 Typical Invocation

Most of these options are not needed for someone wanting to run his own set of tests. The typical way user Biff would run just the Version tests is

linux-box> cd /u/biff/p4/isis/test2

linux-box> perl dfmtest.pl run -m Version

If his tests need a resource file (explained here) he would

use the -r <rsrc\_file> option

linux-box> cd /u/biff/p4/isis/test2

linux-box> ls modules/Biff\*

modules/BiffMod.pm

linux-box> perl dfmtest.pl run -r rsrc.biff -m BiffMod

Ensure resources listed in rsrc.biff exist and DFM knows about them (-z)

linux-box> perl dfmtest.pl run -r rsrc.biff -z -m BiffMod

Run just the modules with the ProvMgr tag

linux-box> perl dfmtest.pl run -r rsrc.biff -T ProvMgr

A few more examples

Run all tests in the Syntax test group of the Acl and AgentHost modules

linux-box> perl dfmtest.pl run -r rsrc.biff -m Acl,AgentHost -g Syntax

Run all tests in the Syntax test groups found in any module

linux-box> perl dfmtest.pl run -r rsrc.biff -g Syntax

Run the VersionUNIX test case in the Version module

linux-box> perl dfmtest.pl run -r rsrc.biff -m Version -t VersionUNIX

Run all the VersionUNIX test cases found in any module

linux-box> perl dfmtest.pl run -r rsrc.biff -t VersionUNIX

Run a module against a remote dfm server using the -h option

linux-box> perl dfmtest.pl run -r rsrc.biff -m API -h mgt-c24

The -z option does a couple of things when used in conjunction with the -r <rsrc\_file> option. When these are used together, ANT will attempt to verify the contents of the <rsrc\_file>, make sure DFM is aware of the resources listed in the file, and for any Scratch objects listed in the <rsrc\_file> ANT will attempt to create a snapshot called ANT\_reset. For aggrs and vols, the snapshot is created on the object. For scratch qtrees, the snapshot is created on the containing volume.

The purpose of these ANT\_reset snapshots is to make it easy for test modules to restore scratch objects to their original state. A test module should endeavor to clean up after itself, including cleaning up the scratch objects it used. ANT provides the scratch\_object\_reset(obj) routine to facilitate the clean up.

4.5 Output From Running Tests

As you can see, the run command supports lots of options. It also creates two log files, both are put in ./logs/<yyyy\_mm\_dd>/ directory. The test.<timestamp>.log file is of more interest. It contains all the information sent to the screen, plus some additional information that can help with debugging test cases.

Originally, each time "perl dfmtest.pl run ..." was run it would overwrite the ./test.log and dfmant.results files. Past versions of these files were not kept around. Now ANT keeps the real files in ./logs/<yyyy\_mm\_dd>/ directory and, on Solaris and Linux, it creates a symlink from test.log and dfmant.results to the latest test.<timestamp>.log and dfmant.<timestamp>.log, respectively. If you want to look at the files from a previous dfmtest.pl run ... you can find them in the./logs/<yyyy\_mm\_dd>/ directory. On solaris and linux, the ./test.log and ./dfmant.results files are symlinks to the latest versions of timestamped files.

Assume Biff invoked the run command three times on Feb 20th.

test.log and dfmant.results are symlinks to the last invocation.

The logs/2006\_02\_20 directory contains files from all 3 runs.

The directory also contains a "current test environment" file for

each of the three runs.

linuxbox> cd /u/biff/p4/isis/test2

linuxbox> ls -l test.log dfmant.results

lrwxrwxrwx 1 biff engr 47 Feb 20 10:51 dfmant.results -> ./logs/2006\_02\_20/dfmant.20060220105147.results

lrwxrwxrwx 1 biff engr 41 Feb 20 10:51 test.log -> ./logs/2006\_02\_20/test.20060220105147.log

linuxbox> ls logs/2006\_02\_20

dfmant.20060220095451.results test.20060220105147.log

dfmant.20060220095538.results test\_env.20060220095451.kalishnikov.htm

dfmant.20060220105147.results test\_env.20060220095538.kalishnikov.htm

test.20060220095451.log test\_env.20060220105147.kalishnikov.htm

test.20060220095538.log

4.6 Getting More Output

DFM ANT now uses log4perl, a popular perl package that facilitates better logging. The full details on how to use log4perl with DFM ANT can be found in the Log4Perl Appendix.

4.7 Debugging

If you've just tried to run some tests and did not get the results you expected then it is debugging time. There are three possible problems; the test environment is not correct for the tests, or the test cases are wrong or broken, or there is a problem with the product. We will walk through some suggestions for determining, and where possible, fixing these problems.

One quick note: If you are debugging existing tests (rather than new ones) then the first thing you may want to do is see some previous results. Of course, you can look through the files in the logs/ directories, but you can also look at our Recent Results web page to check out the most recent official results of existing test cases. Notice that this web page shows results by platform, date, and DFM build. You can look at results by module, too. The best thing to do is look for recent results on the same platform, and if possible, the same DFM release.

If your tests are failing in the same fashion as the recent results then you've hit a known failure. The failure should already be burted. You'd see the burt number on the Test Details page for the test case. If the failure is not already burted then consider burting it if, after triaging the problem, you think it needs burting. If the problem is with the test case then burt it with subtype=dfm\_ant and keyword broken\_test. Here is some helpful info on filing burts.

Remember that the test.log and dfmant.results files can provide valuable clues as to the reasons for failures.

Check the run time environment. The ANT run time environment might not be correct. Try these simple steps to ensure it is correct.

On the command line

> dfm version

Make sure the dfm command returns a version.

If this fails then DFM is probably not installed correctly.

> dfm services list

Make sure all the services are started.

> dfm user list

Make sure you are listed and have full privileges.

> perl dfmtest.pl run -m Version <== Only on Linux/Solaris

> perl dfmtest.pl run -m Version <== Only on Windows

These tests should run. They should all pass. If you get

a "Can't locate NaServer.pm" message then the zapi/ directory is missing

or not setup correctly. Revisit the Setup section.

If all that checks out okay then the test cases may have some special run time setup that is missing. The best way to find this out is to run the tests in the perl debugger. Don't panic, it's not too bad.

Check the test case. A lot of the DFM ANT test cases are easy to duplicate on the command line. You can run the test cases using the perl debugger, as suggested above, or you can read the perl code and try to duplicate the test case on the command line. Just remember that some tests rely on earlier parts of the module to set up conditions for the test. If you do read the code you'll want to take a look at what went on before the test case, too.

Check the product. So you've checked that both the run time environment and the test case is good. What else is there ? Maybe there really is a problem with product! Try to manually do whatever the test case is doing. If you've found a product burt then you should submit the burt. Here is some helpful info on filing burts.

4.8 The Environment File

If you use the -e option when running test cases then the test.env file is populated with information about the run time environment. The file is written as a html file and it consists of two sections. The first section shows info from resource file (if one is specified) and the second part shows info related to the DFM Server and some of the resources.

The thought is that this information may eventually go into a database or otherwise be kept in such a way as we can mine it for information. It could be used to determine what versions of ONTAP did ANT test with, or what types of volumes were used. Things of that nature. Here is an example file.

4.9 Filing Burts

If you are filing a burt on DFM ANT (either infrastructure or tests) then set the subtype=dfm\_ant. If you are filing a product related burt then be sure set the correct type and subtype values. Here is some helpful info on filing burts.

5. Writing Tests

This section discusses the conventions and rules for writing ANT test cases. It does not go over all the support functions supplied by ANT. To get an accurate and up-to-date list of support functions take a look at the online reference pages.

The best way to understand how to write ANT tests is to look at some example code.

5.1 A Simple Test Case

This is a simple test case:

test\_begin('SnmpDeleteDefault');

test\_verify('dfm snmp delete default', 1,

'Error: Cannot delete default snmp community');

test\_end();

The above code should be fairly self explanatory. To indicate the beginning of a test case, the subroutine test\_begin() is called. It takes one argument, which is the name of the test case. The test case is ended by calling test\_end(). These subroutines setup and tear down the environment needed to run a test, respectively. Note: the test case name should not include a colon. ANT uses the colon as a separator/delimitor between the components of the complete name. Same goes for the test\_group name, it should not use colons in the test group name.

The interesting part of the test case is the invocation of test\_verify(). This subroutine takes three arguments: a CLI command, (eg: 'dfm snmp delete default'); the expected return value of the command; and the expected output of the command. The expected output is subject to interpolation as a regular expression, so any special characters must be escaped.

5.2 A More Complex Test Case

If you need to do something more complex than just testing the return value and the output of a given dfm command, then you need to formulate a more complex test case:

test\_begin('UserDeleteInvUser');

test\_body {

my $user = 'dummy';

user\_ensure\_not\_exists($user);

verify\_cmd("dfm user delete $user", 1,

"Error: User $user does not exist.");

user\_verify\_not\_exists($user);

test\_pass();

};

test\_end();

This case begins and ends in a familiar manner, however, it has a much more involved body. The test\_body() subroutine sets up the proper environment and executes the block passed to it. The block should exit with a call to one of test\_pass(), test\_fail(), or test\_skip(). If one of these subroutines is not called, the test will default to the status of SKIP, even if the body was executed successfully.

The subroutines user\_ensure\_not\_exists() and user\_verify\_not\_exists() are defined elsewhere in the test module, and are not part of the standard test or utility subroutines. The other subroutine, verify\_cmd(), is part of the standard set of ANT utility subroutines. verify\_cmd() takes the same arguments astest\_verify(), except that verify\_cmd() must be called within a test\_body().

As you can see, a test case takes one of two forms. Either it is a simple test\_begin(), test\_verify(), test\_end() sequence or it is a more complicated test\_begin(), test\_body {...perl\_code...}, test\_end().

5.3 Test Module Structure

Now let's take a look at how the test module is structured. At the top of your new test module you should include these lines

###########################################################################

#

# $Id: //depot/web/engineering/design-depot/appliance-mgmt/champagne/dfmtest2.mml#38 $

#

# Copyright (c) 2009 NetApp, Inc.

# All rights reserved.

#

###########################################################################

use strict;

Also at the top of the module write a brief description of the purpose of the module. You can also use the AntTag feature on your module. AntTags can be used to select which modules to invoke at during an ANT run. For example, test modules that exercise provisioning manager could have an ANT tag added like this;

# AntTag: ProvMgr

Then at run time, rather than having to list out the modules with a long comma separated list, we can use the -T ProvMgr tag and the modules will automatically be selected.

## This invocation line

#

linxu\_box> perl dfmtest.pl run -r rsrc.A.rtp -m mod1,mod2,mod3,mod4,mod...

## would be replaced by

#

linux\_box> perl dfmtest.pl run -r rsrc.A.rtp -T ProvMgr

As mentioned in the Running Tests section, a complete test case name consists of module\_name:group\_name:test\_name. You can see how this is derived by looking at the example module, Biff.pm, below.

sub BiffTest {

test\_group "Syntax", sub

{

test\_begin('NoArgs');

test\_body {

# test body

};

test\_end();

test\_begin('BadFlags');

test\_body {

# test body

};

test\_end();

};

};

Every module consits of at least one subroutine; it must be named the same as the name of the module, with 'Test' appended. This subroutine is the entry point into the module, and will be called by the harness when the module is loaded. All of the test modules and the harness share a single namespace, however, so you must be careful not to clobber other modules' symbols with your own. The naming convention for subroutines is to prepend the module name to the subroutine; for example, if we wanted to have a subroutine in our example module above, we would call it biff\_something().

Test cases are organized into groups inside of a module. The test\_group() subroutine is used to accomplish this. It takes the name of the group as it's first argument, and an anonymous subroutine as its second argument. The body of the passed subroutine consists of the test cases in the group.

All of these format requirements might sound complicated, but just take a look at some of the existing modules in the modules/ directory and you'll see it is easy to deal with. The Version.pm module is a good one to look at.

Debugging with "print" statements can be achieved in two different ways. We support the elegant Log4Perl method , but we also endorse a simple method. Here's an example of the simple method

sub simpleTest {

# $tstdbg=0 supress debug messages; $tstdbg=1 enable debug messages;

#

my $tstdbg = 0;

$tstdbg = 1 if ( $ENV{TSTDBG} eq "true" );

.

.

.

message("I'm a debug message.\n") if($tstdbg);

If I set the environment variable TSTDBG=true before I run this test then the message(...) will be executed, otherwise it isn't executed.

If I want this extra level of messaging I can set the environment variable. I can use this when debugging the test. I can leave the message() in place so if I need to do debugging at a later time I can again just set the env. var.

5.4 Writing ZAPI tests

Zapi test module file names should be in "z\_dp\_policy\_connection.pm" format. They should start with z\_ and replace the dash in api name with underscores, for example, the test module for "dp-policy-connection" should be called z\_dp\_policy\_connection.pm.

5.4.1 What to include to write ZAPI test

There are two files you will most likely want to include in your zapi test module. lib/Zapi/ZDapi.pm contains lots of useful routines and lib/Zapi ZDapiErrno.pm. contains constants for all the zapi errnos, such as EINVALIDMEMBER and the like. Some where near the top of your zapi test module you'll want to have

• use Zapi::ZDapi

• use Zapi::ZDapiErrno

5.4.2 Sample code

sub z\_rbac\_admin\_role\_addTest {

...

# Global for all tests

### Get the dfm server

my $dfm\_server = get\_dfm\_server();

### Create dfm zapi object

my $dApi = Zapi::ZDapi->new( $dfm\_server, $user, $password );

### Set api name to rbac-admin-role-add

$dApi->set\_api\_name("rbac-admin-role-add");

...

### You can create test group for different testing area.

test\_group "GroupName", sub {

test\_begin('TestName');

test\_describe("Test description.");

test\_body {

# Test

...

### Indicate test pass

test\_pass();

}; # end test body

test\_end();

### Real example from z\_rbac\_admin\_role\_add.pm

### Case one, negative

test\_begin('HasFakeTag');

test\_describe("Verify api with fake tag should fail.");

test\_body {

### Set up xml tag in the hashtable

my %param = (

"admin-name-or-id" => $testUser,

"role-name-or-id" => $testRole,

$fakeTag => $fakeTag,

);

### Set the param to api

$dApi->set\_api\_param(\%param);

### Run api and compare return code

$dApi->validate\_return\_code(ZapiErrno::EINVAL);

test\_pass();

}; # end test body

test\_end();

### Case two, positive

test\_begin('addRoleIdToUserId,addSameRoleIdToUserId');

test\_describe("1. Verify add role to user using id should succeed." .

"2. Verify add the same role using id should fail.");

test\_body {

# Remove all roles from user

deleteAllRolesFromUser($testUser);

my $rbacRole = Rbac::ZRbacRole->new($dApi);

my $userId = $rbacRole->get\_admin\_info\_list\_admin\_id($testUser);

my $userRoleId = $rbacRole->get\_info\_list\_role\_id($testRole);

# use id instead of name

my %param = (

"admin-name-or-id" => $userId,

"role-name-or-id" => $userRoleId,

);

### Set param

$dApi->set\_api\_param(\%param);

### Get hash output

my $resultHash = $dApi->get\_hash\_output();

### Get amdin name from output

my $returnName = $resultHash->{"results"}->

{"rbac-admin-name-or-id"}->{"admin-name"};

if($testUser ne $returnName){

### Fail the test

test\_fail("Got $returnName, but expected $testUser");

}

# Assign the same role to the user again, it should

# return proper error

$dApi->validate\_return\_code(EROLEASSIGNED);

test\_pass();

}; # end test body

test\_end();

}; ### end test group

} # end sub z\_rbac\_admin\_role\_addTest

• Using ZDapi routines

• new ZDapi object, this sets server credential

• my $dfm\_server = get\_dfm\_server();

• my $s = Zapi::ZDapi->new( $dfm\_server, $user, $password );

• set api name, so it would build the rbac-operation-delete tag

• $s->set\_api\_name("rbac-operation-delete");

This sets xml tag like this

<rbac-operation-delete>

...

</rbac-operation-delete>

• initialize %param, pass all xml tag as key-value pair

• my %param = (

• "admin-name-or-id" => $testUser,

• "role-name-or-id" => $testRole,

• $fakeTag => $fakeTag,

• );

This builds xml tag like this

<admin-name-or-id>testUserValue</admin-name-or-id>

<role-name-or-id>testRoleValue</role-name-or-id>

<fakeTagValue>fakeTagValue</fakeTagValue>

• set api param

• $s->set\_api\_param(\%param);

• execute api, there are three methods to use

• Runs the api then compares the return code with the code provided.

• $s->validate\_return\_code(ENOTFOUNDOPERATION);

• Runs api and makes sure return code is 0, a success. Then returns a reference to a NaElement structure. You can walk through the structure to get the children and its content.

• $output = $s->run\_api();

• Runs api and makes sure it is a success, then return a result hash. It actually calls run\_api(), parse its result to a hash, then return the hash.

• $result = $s->get\_hash\_output();

• If you want to see the output in raw xml, you can use

• $result = $s->get\_hash\_output({"print"=>"yes");

It would print out the xml and some debug info.

• Getting the results

• Get a value from hash

The hash is using xml tag as key, so check the zapi doc for its tag. For example, the output of rbac-access-check api contains global-usergroup-status

$globalGroupStatus = $hashOutput->{results}->{"global-usergroup-status"};

• Get a value from a hash array from rbac-role-admin-info-list

• %param = ( "admin-name-or-id"=>"$user",

• "follow-role-inheritance"=>"true",

• );

• $s->set\_api\_param(\%param);

• $p = $s->get\_hash\_ouput("rbac-role-admin-info-list");

• $roleId = $p->{results}->{"role-list"}->{"rbac-role-resource"}->[0]->{"rbac-role-id"});

• In addition to the routines in lib/Zapi/ZDapi.pm the Reference Pages have other useful routines.

• 5.5 Best Practices for Writing ANT Test Cases

• The following items are considered best practices for writing DFM ANT test cases.

• Utilize the resource file to gather login data for a filer. The resource file mechanism is the right way to specify filers, their login credentials and other pertinent information. Review the current resource file template (rsrc.template) to determine what information can be gathered. Do not hard-code arbitrary data such as password/login information. If the information you require is not in the resource file, consider updating the resource file to add the information you require.

• Use dfmtest::test\_describe() to indicate what your test is trying to verify. As others may follow your work, this helps test developers understand your original intention and will help them update your test case \*when\* such a need arises.

• The output format of dfm commands can change from one release to the next. If a test or library relies on a certain output format of a DFM command, then the code should either use the -V n option or ask for the output in perl format (and handle the perl format). A command may have gone through any number of output format changes so when you write your code you'll need to figure out the correct format number and use that. Here is an example of how the output format can change. Notice the only difference in the two commands is the number after the -V option, but the difference in the output format is significant.

• mgt-ln13> dfpm dataset list -m -V 8 ant\_dataset\_1

• Id Node Name Type Name

• ---- -------------------- --------------- ----------------------------------------

• 22389 Mirror volume mgt-u71:/ant\_dataset\_0\_backup\_26

•

• mgt-ln13> dfpm dataset list -m -V 9 ant\_dataset\_1

• Id Node Name Dataset Id Dataset Name Type Name

• ---------- -------------------- ---------- ------------- -------- -----------------------------

• 22389 Mirror 22396 ant\_dataset\_1 volume mgt-u71:/ant\_dataset\_0\_backup\_26

If you write code that expects output to be formatted like the -V 8 version, but you actually get the -V 9 format, most likely your code will mishandle the output. That's why if you use the dfm cli and parse it's output you should be sure to use the -V n. Here's an example

($status, $output) = run\_cmd("dfpm dataset list -m -V 8 $my\_ds");

code\_to\_parse\_$output

The other option is to use the -F perl option for the dfm cli and then process that. You can see an example of how to do this in the user\_ensure\_exists routine of modules/User.pm module.

• Do not check in files containing control-Ms.

• Do not create temporary files in the test2/ directory. If tests need to create temp files, they can test2/data or test2/config directories.

• Use double quotes whenever possible. We've seen some problems on Windows when a string contains single quotes. It's more reliable to use double quotes for strings.

• ### Do this

• verify\_cmd("dfpm policy list \"Backup and Mirror\" ", 0);

•

• ### Not this

• verify\_cmd("dfpm policy list 'Backup and Mirror' ", 0);

• The test module should create its required test environment, then clean it up. Ensure that you leave the test environment as you found it. Also, if the test environment must exisit in a pariticular state, it is your responsibility to create that state (and revert it back to its original state).

• Call ALL custom routines from within the subroutine <module\_name>Test. All helper routines (i.e. module\_setup(), module\_cleanup() ) should be called from within the subroutine <module\_name>Test. Failure to do so could result in disruption to the automated nightly test.

• Use dfmtest::module\_abort() to halt execution of your test module. If you must call die() outside of a test case, use module\_abort() instead. module\_abort() will provide some context that will be useful for debugging purposes.

• Never call test\_fail() without providing a reason (alternatively, always use errmsg\_wrong\_output()/errmsg\_wrong\_status() ) A test that fails should make every effort to communicate the reasons why. test\_fail() takes a simple string argument. The convenience methods errmsg\_wrong\_output() and errmsg\_wrong\_status() should almost always be used to format your message to test\_fail.

• Always prepend the module name to all subroutine declarations. Fully qualified names aren't used within dfmant. As such, it is important to utilize this naming convention to avoid collisions in the namespace.

• Test cases should verify that necessary preconditions are met before executing. In other words, ensure that the necessary test environment exists before running your test. If not, please call test\_skip() (preferrably with a reason for doing so...).

• Test cases should not rely on the results of other test cases. Test cases should run independently of one another. If dependencies exist between test cases, those should be checked from within the test case before execution.

• If your test cases need new libraries, config files, data files or such, then make sure you add those files in the appropriate location. Take a look at test2/ appendix for guidance.

• If your test cases need a special package, such as something from cpan.org, then you need to make sure it is available on all the platforms (Windows, Solaris, RedHat, Suse). Make sure the libraries exist on Windows as well as on the Solaris and Linux environments.

• Whenever possible, tests should use the DFM ID when dealing with ossv hosts and netapp host agents. The reason for this is that often times a single host can be both an ossv host and a netapp host agent. So if a test tries to use the host name, dfm may complain about an ambigous name.

• On linux, it is usually a bad idea to run ant tests as root. Customers don't run as root so the tests shouldn't do that either. ant has the run\_cmd() routine to allow individual commands to be run as root.

• Never code "forever" loops in your test cases. Use a timer on loops that may otherwise loop forever. For example, if a test is waiting for a dataset to go conformant, make sure there is a timer. If the dataset doesn't go conformant within a reasonable time, exit the loop.

• If your test module does some setup, put the setup in a test case. If the setup test case fails, have the appropriate set of subsequent test cases skip. Use the skip\_group\_if\_test\_fails() routine. See modules/DPDataset.pm for an example.

• Changing dfm, dfpm, dfdrm, and dfbm output can break ant tests. Whom ever changes the output is responsible for fixing all the tests that are broken by the change.

• Utilize PERL best practices when writing code (Recommendations come from Perl Best Practices, by Damian Conway.

• If there are many ways of writing code, chose the option that is most readable

• Always return explicitly, not implicitly from any function

• Always use the 'strict' pragma and declare all variables using the "my" keyword

• Favor the use of underscores as a separator in variable names

• Do not unecessarily abbreviate variable names at the expense of clarity. Include the first portion of a name (with at least one vowel) if your are abbreviating (e.g. temp is better than tmp)

• Use "ref" as part of the variable name when referring to references (e.g. $filers\_ref)

• Use the plural form of a noun for array variable names (e.g. @filers rather than @filer)

• use PerlTidy to format your code (http://perltidy.sourceforge.net).

5.6 The Resource File

In August, 2005 we added the resource file option (-r ) to ANT. We found that lots of test modules had hardcoded values for certain resources. Additionally, some sets of modules had their own configuration/resource file. This made running DFM ANT in different environments and with different sets of resources (filers, netcache, SRM Hosts, etc) impossible. To solve this problem we have made a couple changes to DFM ANT. We replaced the hardcoded values in the test modules with calls to testutil::dfmant\_get\_object(). We also got rid of the other configuration/resource files. Test modules now get their resources using dfmant\_get\_object() or dfmant\_get\_objects(). When a run of dfm is started ('perl dfmtest.pl run...') if a resource file is specified then it is pulled into the run environment. Tests use dfmant\_get\_object() routine to get their desired resources.

An example will help explain this.

The old version of modules/BMAcl.pm had these hardcoded values

my $sechost = "fucilli";

my $secvol = "fucilli:/alfredo;

The new version of modules/BMAcl.pm has

my ($sechost, $secvol) = dfmant\_get\_object("sechost", "volcnt=1");

The actual values of the host and volume are specified in the resource file. We could have a Sunnyvale resource file and a BTC resource file.

rsrc.A.svl file could have

SVSecHost: fucilli.lab.netapp.com

SVSecVols:

- fucilli:/marinara

- fucilli:/alfredo

while the rsrc.A.nb file could have

SVSecHost: chi.btc.netapp.com

SVSecVols:

- chi:/cold

- chi:/warm

This allows the same tests, BMAcl tests in this case, to be run in different environments and using different resources. Take a look at rsrc.template for a complete list of what can be specified in a resource file. The dfmant\_get\_object() entry in the reference manual explains how to access the resources.

Get the full details on the resource file and its relationship to dfmant\_get\_object() and dfmant\_get\_objects() take a look at dfmant\_get\_object()reference page.

Verifying objects listed in the resource file can now be done using the -z option to the ANT run command. The -z option will cause ANT to check as much of the resource file entries as possible. It checks that the objects exist and that DFM is aware of them.

Here's an example of using the -z option with the rsrc.A.svl resource file. You can see that it gives some status information, as well as some warnings on things it could not do, like add mgt-pc24 as a NHA.

mgt-ln13.lab.netapp.com> perl dfmtest.pl run -r rsrc.A.svl -z -m Version

=== Welcome to DFM ANT ===

Running on Linux

Resource file is rsrc.A.svl

Resource Check: [INFO]: dfm already knows about mgt-reg-u10.rtp.netapp.com

Resource Check: [INFO]: set login credentials for mgt-reg-u10.rtp.netapp.com

Resource Check: [INFO]: set NDMP login credentials for mgt-reg-u10.rtp.netapp.com

Resource Check: [INFO]: dfm already knows about remington.lab.netapp.com

Resource Check: [INFO]: set login credentials for remington.lab.netapp.com

Resource Check: [INFO]: set NDMP login credentials for remington.lab.netapp.com

Resource Check: [INFO]: dfm already knows about winchester.lab.netapp.com

Resource Check: [INFO]: set login credentials for winchester.lab.netapp.com

Resource Check: [INFO]: set NDMP login credentials for winchester.lab.netapp.com

Resource Check: [INFO]: dfm already knows about mgt-vf060

Resource Check: [INFO]: set login credentials for mgt-vf060

Resource Check: [INFO]: dfm already knows about mgt-vf061

Resource Check: [INFO]: set login credentials for mgt-vf061

Resource Check: [INFO]: dfm already knows about mgt-vf062

Resource Check: [INFO]: set login credentials for mgt-vf062

Resource Check: [INFO]: dfm already knows about mgt-vf063

Resource Check: [INFO]: set login credentials for mgt-vf063

Resource Check: [INFO]: dfm host set mgt-c23.lab.netapp.com hostLogin= hostPassword=

Resource Check: [INFO]: dfm host set mgt-ln07.lab.netapp.com hostLogin=admin hostPassword=private

Resource Check: [INFO]: dfbm primary host modify -U test -P 1testtest 67113

Resource Check: [WARN]: Could not add mgt-pc24.lab.netapp.com as agent !!

Host: mgt-ln13

User: marlon

DFM version: 3.8

DFM dir: /opt/NTAPdfm

Running tests...

Module Version started at Mon Jun 22 17:14:56 2009

Version:List:VersionUNIX ............................................. PASS

Version:List:VersionList ............................................. PASS

Test Results:

PASS: 2

DFM ANT will not quit or error out if it is unable to validate all the resources. Instead, it will issue a warning message and continue with the testing. It seems too onerous to mandate a perfect environment, especially since not every invocation of ANT will run the full set of test modules.

You don't need to use the -z option every time you run ANT. You really only need to use it the first time you run ANT with a particular resource file and a particular version of DFM.

5.7 Adding To The Resource File

Before taking the steps listed below, make sure the resource you need is not already part of the resource file. Take a look at the rsrc.template file in this directory. It is both a template, used to create customized resource files, and a complete list of all the resources currently supported by the resource file mechanism.

There are three places where you will need to make changes in order to add a resource. The first place is the rsrc.template file. You'll need to figure out where the resource should go. Second, you'll need to add the helper routines to testutils.pl and, third, you'll need to call dfmant\_get\_object() or dfmant\_get\_objects() in your test modules.

The format of the resource file is fairly simple but if you want details on it you can visit the YAML web site, http://www.yaml.org. The basic YAML syntax that is used in the resources files is one of these three formats

Key\_1: A String Value

Key\_2: [A, comma, separated, list]

Key\_3:

- A

- list

- of

- items. One item per indented dash.

In perl these three keys:value pairs would look like

$hash{"Key\_1"} = "A String Value";

$hash{"Key\_2"} =

"A" "comma" "separated" "list"

$has{"Key\_3"} =

"A" "list" "of" "items. One item per indented dash."

Assuming you understand the format, you now need to decide what the key will be and where it will fit. If it is a resource that is associated with a filer then it should probably be part of the filer list. If it is independent of other resources then it can be on its own. Note that while you are developing your new module and adding the new resource you can create a very simple resource file with just the resources you need.

%> perl dfmtest.pl run -r my-new-rsrc -m MyNewModule

my-new-rsrc has just the resources needed for MyNewModule. When you are ready to check in your test module and these new-resource related changes you'll need to make sure the rsrc.template file is updated.

In order to provide the resource to your test modules, you will need to write the helper function that is called by dfmant\_get\_objects(). You'll also need to update dfmant\_get\_objects() routine to call your new helper function when the resource is requested. Don't forget to keep the POD comments above dfmant\_get\_object routine up to date. Lastly, you'll need to update the keys at the top of the testutil.pl file. You'll know best where to add the calls to dfmant\_get\_object() within your test module. If you need examples of this, take a look at modules/BMSecHost.pm.

5.8 Using Scratch Objects In Your Tests

As we get more and more test modules added to ANT, the overall run time grows, too. It used to take just a couple hours to run all the ANT tests. Now it can take around 24 hours. A good part of the time is spent by tests that create new objects on the filers and then wait for DFM to discover those objects. This option is still available to test modules, but a new ANT feature is the preferred way because it does not require the discovery time.

ANT now supports the notion of ready-to-use scratch objects. These objects are specified in the resource file and are associated with a specific filer. Supported object types are aggregates, non-larger aggregates, larger aggregates, traditional volumes, flexible volumes and qtrees. Tradvols, flexvols and qtrees are supported on vFilers, too.

The agreement between ANT and you (the tester) is that any objects you list as scratch objects in the resource file must exist on the filer before you start running ANT. For example, if my resource file says winchester:/ANT\_TRADVOL is a scratch tradvol then I should make sure that traditional volume really does exist on winchester. The other part of the agreement is that if a test module uses a scratch object, the module will clean up and reset that object. This is pretty much the same agreement that has already in place for other resources listed in a resource file.

Scratch objects can be requested using the dfmant\_get\_objects() routine. POD documentation can be found here but let's take a look at a couple examples right now.

# Request 2 scratch qtrees that reside on a vFiler

#

my %hash = (

'filerType' => "vFiler"

);

my (@s\_obj ) = dfmant\_get\_objects('scratchqtree', 2, \%hash);

# Request a NonLarger Aggregate that resides on a filer with SV Sec License,

# and the filer holding the NonLA isn't called remington or colt.

#

%hash = (

'type' => "NonLA",

'SVSecLicense' => 1,

'notFilers' => "remington,colt"

);

(@s\_obj ) = dfmant\_get\_objects('scratchaggr', 1, \%hash);

# Request 2 flex vols on a filer with SnapMirror license and then

# request an aggregate from a filer with SnapMirror licnense, but is not

# one of the filers return for the 2 flex vols.

#

%hash = ('

SMLicense' => 1

);

my (@s\_vols) = dfmant\_get\_objects('scratchfvol', 2, \%hash);

my $notlist = extract\_filernames(@s\_vols);

%hash = (

'SMLicense' => 1,

'notFilers' => "$notlist"

);

my (@s\_aggr) = dfmant\_get\_objects('scratchaggr', 1, \%hash);

Only scratch objects listed in the resource file will be considered for these requests.

6. FAQ

Include stuff about publishing nightly results

Q: I have a variable that I defined globally in my module, and it disappears on me!

A: If you are defining your variable using my, this is a scoping bug

in Perl versions 5.6 and greater (It has been verified on 5.6.x and

5.8.0). As a workaround use 'our' to define your global variable. For this

reason, and to avoid polluting the shared namespace, it is recommended

that you do not use global variables if possible.

7. Appendix A: Perl Debugger

The perl debugger is fairly easy to use and has a few simple commands that are all you usually need. To invoke the debugger you supply the -d flag as the very first argument to perl. To get more info on the the perl debugger you can run man perldebugger or invoke the perl debugger and then issue a h.

The perl debugger has a lot of features. A few of the most common features are listed here.

Option Description

l Display the 10 lines at the current point of execution.

l subroutine Display the first 10 lines in subroutine.

n Next, steps over subroutines.

s Take a step in execution. Will step into subroutines.

b [line|subroutine] Set a breakpoint at line or at the beginning of subroutine.

c Contine to next breakpoint or end of execution.

c subroutine Acts like a one time break point. Execution continues until subroutine.

p var\_name Prints the value of a variable.

T Prints stack trace.

One thing that is a little odd about DFM ANT is the way modules get executed. The test module is executed via a perl eval{..} statement so the modules code is not part of the run time environment until it is pulled in by the eval statement. You cannot set breakpoints in a test module until it is loaded by the eval statment. This adds a couple steps to the debugging process, but it not too bad. Here's a sample debugging session. (note that the line numbers you see and exact code may not match the example below). Our goal is to find the line with the eval that picks out our function. Annotations are instigated by ###.

kalishnikov> perl -d dfmtest.pl run -m Version ### Using '-d' invokes debugger

Default die handler restored.

Loading DB routines from perl5db.pl version 1.07

Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(dfmtest.pl:26): require 'testutil.pl';

DB<1> b \_test\_module ### Set breakpoint at \_test\_module()

### The 'eval {...}' is inside \_test\_module()

DB<2> c ### Go ahead with running.

=== Welcome to DFM ANT ===

Running on Solaris

Host: kalishnikov

User: marlon

DFM version: Rmarlon\_060223\_1600

DFM dir: /opt/NTAPdfm

Running tests...

main::\_test\_module(dfmtest.pl:696): $current\_module = $\_[0];

DB<2> l ### The 'l' displays 10 lines, including breakpoint at 696.

###

696==>b $current\_module = $\_[0];

697

698: foreach my $pattern (@disabled\_patterns) {

699: if ( "$current\_module:" =~ m[^$pattern$] ) {

700: return;

701 }

702 }

703

704: my $function = $current\_module . 'Test';

705

DB<3> l 704-713

704 my $function = $current\_module . 'Test';

705

706: require "$module\_dir/$current\_module.pm";

707

708 #

709 # We need to turn of strict refs so we can call subroutines

710 # with strings.

711 #

712: no strict 'refs';

713: eval { &$function(@module\_args) };

DB<4> c 713 ### Continue until hit the eval {...}

main::\_test\_module(dfmtest.pl:713): eval { &$function(@module\_args) };

DB<3> p $function ### Check function name

VersionTest

DB<5> s

main::\_test\_module(dfmtest.pl:713): eval { &$function(@module\_args) };

DB<5> s ### Now in Version.pm. Can set breakpoint in it.

main::VersionTest(modules/Version.pm:75):

75: };

DB<6> l 50-60

50 '^dfm\s+\d+\.\d+ \([\w\.]+\)');

51: test\_end();

52 }

53

54: test\_begin('VersionList');

55 test\_body {

56: my ($status, $output) = run\_cmd('dfm version');

57: if (0 != $status) {

58: test\_fail(errmsg\_wrong\_status(0, $status, $output));

59 }

60: my @lines = split "\n", $output;

DB<7> c 54

Version:List:VersionUNIX ............................................. PASS

main::CODE(0x9957e0)(modules/Version.pm:54):

54: test\_begin('VersionList');

55: test\_body {

DB<8> T

. = main::\_\_ANON\_\_[modules/Version.pm:75] called from file `dfmtest.pl' line 738

. = eval {...} called from file `dfmtest.pl' line 738

. = main::test\_group('List', ref(CODE)) called from file `modules/Version.pm' line 75

. = main::VersionTest() called from file `dfmtest.pl' line 713

. = eval {...} called from file `dfmtest.pl' line 713

. = main::test\_module('Version') called from file `dfmtest.pl' line 1046

. = main::run\_tests('Version') called from file `dfmtest.pl' line 1425

DB<8> c

Version:List:VersionList ............................................. PASS

Test Results:

PASS: 2

Debugged program terminated. Use q to quit or R to restart,

use O inhibit\_exit to avoid stopping after program termination,

h q, h R or h O to get additional info.

DB<8> q

kalishnikov>

While this debugging session does not actually debug or fix anything, it does illustrate how simple it is to use the perl debugger.

One more note; On some systems you may see an error list this when you try to run the perl debugger. To get around it, use this setting.

The error message looks like:

Cannot do `initialize' in Term::ReadLine::Gnu at /usr/local/lib/perl5/5.6.1/perl5db.pl line 1767

Log::Log4perl::CODE(0x88f37d4)(lib/Log/Log4perl.pm:5):

The solution is to do this before invoking the debugger

linuxbox> export "PERL\_RL=Perl o=0"

8. Appendix B: Eclipse Setup

To\_be\_Added.

9. Appendix C: Nightly Results

During RRT, FFT and even before FFT we (the QA Team) set up a number of systems where we run the full set of DFM ANT tests diurnally. For Janneau we ran the full set of DFM ANT test cases on five different platforms. As part of the nightly process the results get logged to a database and are presented through a web interface. The Recent Results web page shows how the nightly runs are doing.

10. Appendix D: test2/ layout

The top level of DFM ANT is the test2/ directory. Inside test2/ you will find the following files and directories.

Dir. or File Descrption

bin/ Holds executables used by ANT.

config/ Holds config files that are used or generated by test modules.

data/ Holds data files that are used or generated by test modules.

dfmant/ Holds code used to generate Recent Results web pages.

dfmant.results While be a symlink to the latest test\_message() output from running ANT.

dfmtest.pl Perl script used to launch DFM ANT tests.

disabled\_patterns.txt/ Holds a set of regular expressions are used to determine what tests NOT to run.

disabled\_tests.txt Holds a list of test cases NOT to run.

lib/ Holds libraries used by DFM ANT.

logs/ Holds all the logs from previous runs of DFM ANT.

modules/ Holds all the test modules.

rsrc.template The template (and example) resource file.

test.env Is a symlink to the real test environment file found under logs/. This get created by using the -e option.

test.log Is a symlink to the real test log file found under logs/.

utils/ Holds some utilities used by DFM ANT.

If you are writing tests that need data files, or config files or require new, special libraries then you should put those files in the appropriate directory.

11. Appendix E: DFMANTDOC

The DFMANTDOC contains a complete list of DFM ANT test modules and their test groups and test cases.

12. Appendix F: Better Debugging Using Log4Perl

Log4Perl provides a means to control output. It provides a couple different ways to control output. You can set levels, categories and appenders. I will not go over all the details of log4perl, but rather show you how you can use it within DFM ANT. See log4perl on cpan.org for details on log4perl. Also, "Retire your debugger", by Michael Schilli, is a good intro to log4perl.

Why might you care about this doodad? Because log4perl can help with debugging your perl code. You add some "logging" code to your perl files and then you can use log4perl configuration files to turn on and turn off the logging code. Reality is always better than theory for understanding this stuff so let's look at some real files.

Here is the default log4perl log file used by ANT.

### $Id: //depot/web/engineering/design-depot/appliance-mgmt/champagne/dfmtest2.mml#38 $

###

### This file is used for normal, happy DFM ANT nightly runs.

### It controls how much output is generated and where the output

### is sent. It is set up to generate the traditional amount of

### output to the traditional locations. If you want to do module

### debugging then you should copy this file and edit that copy, rather

### than changing this file. You can use the "-c " run option

### to tell ANT to use your log4perl config file, rather than this one.

###

### The DFM ANT User's Guide for more details on how to use log4perl

### in the ANT world. The guide is located at

### http://web.netapp.com/engineering/design-depot/appliance-mgmt/champagne/dfmtest2.html

###

### This section causes the traditional output to be sent to the

### traditional locations; the screen and the log file.

###

log4perl.logger = INFO, FileApp, ScreenApp

log4perl.appender.FileApp = Log::Log4perl::Appender::File

log4perl.appender.FileApp.filename = sub { ant\_logfile(); };

log4perl.appender.FileApp.layout = PatternLayout

log4perl.appender.FileApp.layout.ConversionPattern = %m

log4perl.appender.FileApp.mode = append

log4perl.appender.ScreenApp = Log::Log4perl::Appender::Screen

log4perl.appender.ScreenApp.layout = PatternLayout

log4perl.appender.ScreenApp.layout.ConversionPattern = %m

log4perl.appender.ScreenApp.Threshold = WARN

###

### To get some extra debugging info in the log file you can uncomment

### entries in this section. The last part of the name corresponds to the

### infrastructure routine that will generate the additional debugging

### information. For example, if you uncomment the log4perl.logger.run\_cmd=DEBUG

### line then >>> every call <<< to run\_cmd() will generate debugging info.

###

# log4perl.logger.run\_cmd=DEBUG

### Add your dynamic logging entries below here.

### Please do not check in this file with your changes.

log4perl supports five levels; from lowest to highest they are DEBUG, INFO, WARN, ERROR, and FATAL. You set a log level, then anything at that level or higher will get logged to the appropriate appenders. The line "log4perl.logger = INFO, FileApp, ScreenApp" sets the logging level for the FileApp and ScreenApp appenders to INFO. Any log request (shown next) of levels INFO, WARN, ERROR or FATAL will be logged to the appenders. Think of appenders as where the log info gets sent. In this case the appenders are the screen and the ANT log file. The line "log4perl.appender.ScreenApp.Threshold = WARN" adjusts the logging for the screen to be WARN and higher. After this adjustment, INFO and above will go to the FileApp appender (this is the ./logs/<yyyy\_mm\_dd>/test.<timestamp>.log file) and WARN and above will go to the ScreenApp appender (the screen). This is the standard, typical behavior for ANT.

Take a look at the line "# log4perl.logger.run\_cmd=DEBUG". This line is currently commented out so log4perl ignores it. If it were uncommented then it would tell log4perl there is a new log4perl category, run\_cmd, and its log level is set to DEBUG. If you are adding log4perl to your code then you will create a new category. Typically, the category is named the same as the routine or file it relates to.

Uncommenting the line will tell log4perl there's a category called run\_cmd with a level of DEBUG. To make use of this category, we added this code snippet inside the subroutine run\_cmd() (in testutil.pl).

my $logger = Log::Log4perl->get\_logger("run\_cmd");

$logger->debug("\nDEBUG: run\_cmd() status and output");

$logger->debug("\nDEBUG: +++ STATUS = $status +++");

$logger->debug("\nDEBUG: +++ OUTPUT\_BEGIN +++\n$output \n+++ OUTPUT\_END+++\n");

This code will generate output only when category run\_cmd is defined. If the line "# log4perl.logger.run\_cmd=DEBUG" is uncommented then the category is defined and any subsequent run of ANT will generate output like these when the run\_cmd() subroutine is called. Here's sample output from when the run\_cmd category is defined.

Run command: "dfm version dfm"

DEBUG: run\_cmd() status and output

DEBUG: +++ STATUS = 0 +++

DEBUG: +++ OUTPUT\_BEGIN +++

dfm 3.6 (RchampagneN\_071027\_1130)

But here does this output go ? Since the ScreenApp appender has its threshold set to WARN the output will not go to the screen. However, the output will go to the FileApp appender because its threshold was not set.

For you to use log4perl in your code you will need to do a couple simple things. First, figure out what to call the category. Usually it is named after the subroutine or file it relates to. Then, in your file you will need to add a call to get the logger for the category and you'll need to add calls to log whatever info you want logged. Lastly, you should your new category to the ant.logger.conf file, but make sure it is commented out.

An example, I want to add some debugging messages to the Version.pm test module. In modules/Version.pm I add

my $version\_logger = Log::Log4perl->get\_logger("Version");

Then in places in the Version.pm test cases I add lines like

$version\_logger->debug("DEBUG: $version\n");

I then copy ant.logger.conf to marlon.logger.conf. I edit marlon.logger.conf and add this line at the end of the file

# Category for modules/Version.pm debugging

log4perl.logger.Version=DEBUG

I then run DFM ANT with my logger file

linuxbox> perl dfmtest.pl run -c marlon.logger.conf -r rsrc.ant.svl -m Version

I check the output in the test log file and make sure I'm seeing the debugging messages I expected. If all that looks good, I then add the two lines, bothcommented out, to ant.logger.conf and check it in, along with the updated Version.pm file.

13. Appendix G: Setting Up Official ANT runs

Once we start a new DFM project we will eventually want to run DFM ANT against the DFM Server. This appendix describes the steps necessary to set up the official ANT run for a DFM project. Projects typically get their own perforce codeline or branch. I will use both codeline and project to refer to the DFM project.

I'm going to assume that your target test host has perforce and perl already installed.

a. You should create a burt for the perforces changes you'll make. Burt 299781 is the burt I have used for Isis-related changes.

b. On solaris and linux we typically run the ANT tests as user ctbuild. On Windows we typically run them as Administrator.

c. You should identify on what system(s) you will run ANT and when you will run it. On solaris and linux systems you can use crontab -e to schedule ANT runs. On Windows you can use the Schedule Tasks mechanism. You can log into mgt-reg-ln01.rtp.netapp.com as ctbuild and run crontab -e to see a Linux example. Solaris looks just the same. You can remote-desktop to mgt-reg-pc01.rtp.netapp.com as Administrator and look at the schedule tasks to see a Windows example.

d. Set up a perforce workspace and fix the dfminst.pl file. Follow the steps in the ANT Setup to create a perforce workspace. Check dfminst.pl file (in the test2/ directory) and make sure it will install the correct version of the DFM Server. Check that GET ... lines in subroutines get\_special\_build() and wget\_file() use the correct path for your codeline. Make the appropriate changes and p4 submit the file. Do this change as yourself, not ctbuild.

e. Set up the linux and solaris ANT runs. The steps for linux and solaris are the same.

a. Log into the linxu/solaris host as ctbuild, or whatever user you plan to use, and create a perforce workspace.

b. Copy run\_dfmant\_nix to run\_dfmant and make the edits suggested in that file.

c. Make the output/ and logs/ directories.

d. Set up scheduled runs using crontab -e. The typical entry looks like

e. 5 18 \* \* 3 cd /u/ctbuild/p4/isis-sus/test2 && ./run\_dfmant > run\_dfmant.output 2>&1

This says to execute run\_dfmant every Wednesday at 6:05PM.

f. You may want run ANT once manually before the cron job gets kicked off. This is so you can do a quick sanity check that everything is set up correctly.

f. Set up the Windows ANT runs.

a. Log into the Windows host as Administrator and create a perforce workspace. Follow the steps in the Windows section of ANT Setup.

b. Copy run\_dfmant\_win1 to run\_dfmant\_part1.pl and make edits suggested in that file.

c. Copy run\_dfmant\_win2 to run\_dfmant\_part2.pl and make edits suggested in that file.

d. Create logs/ folder at the top level of the p4 client.

e. Set up the scheduled tasks. There are three tasks to schedule for each run of ANT. The order of the tasks are

a. run run\_dfmant\_win1

b. reboot host

c. run run\_dfmant\_win2

Each of these needs to be a Scheduled Task. Here are the properties for one example.

Run: C:\Perl\bin\perl.exe run\_dfmant\_part1.pl

Start In: C:\baldi-win

Run as: MGT-REG-PC01\Administrator

Schedule: Weekly at 6:00PM every Monday

Run: C:\WINDOWS\system32\shutdown.exe -r -f

Start In: C:\WINDOWS\system32

Run as: MGT-REG-PC01\Administrator

Schedule: Weekly at 7:15PM every Monday

Run: C:\Perl\bin\perl.exe run\_dfmant\_part2.pl

Start In: C:\baldi-win

Run as: MGT-REG-PC01\Administrator

Schedule: Weekly at 7:25PM every Monday

f. You may want run ANT once manually before the schedule tasks get kicked off. This is so you can do a quick sanity check that everything is set up correctly.

14. Appendix H: Example of Run Time Env. File

Here's an example of what the test environment file looks like. This file is created when the -e option is used with the "run" command. This example shows the contents of the resource file that was used during the run, and it shows the actual resources in place at run time.

Test Env Data

============RESOURCE FILE============

--- HostSystems

---

- !perl/harness::HostSystem

Name: mgt-c23.lab.netapp.com

OS: solaris

Software:

- HostAgent:

ManageCred: admin:private

MonitorCred: guest:public

OS: solaris

Plugins:

- SRM

- SAN

Port: 4092

ReleasePath: /rlse/Rcaffeine2.3.2xN/release

big\_path: /u/ctbuild/srmC-bigpath

small\_path: /u/ctbuild/srmC-bigpath/h

- HostAgent: {}

OSSV:

NDMPCred: test:1testtest

SVPirHost: mgt-c23.lab.netapp.com

SVPriDirs:

- mgt-c23:/ossvC-01

- mgt-c23:/ossvC-02

- mgt-c23:/ossvC-03

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: mgt-ln05.lab.netapp.com

OS: linux

Software:

- DFM:

GlobalFullControlCred: root:sundance

Licenses:

- core:

- non: HHLVBTQFEFPBBG

- add: BIOVBTQFEFPBBG

- dataprotection:

- non: JYMDITQFEFPBBG

- add: XNOICYBAZOOBBG

- provisioning:

- non: VSKSITQFEFPBBG

- add: JIMXCYBAZOOBBG

- disasterrecovery:

- non: THGWJTQFEFPBBG

- add: HXHBEYBAZOOBBG

LoginCred:

- testu1:testp1

- ctbuild:11ctbuild

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: mgt-ln06.lab.netapp.com

OS: linux

Software:

- DFM:

GlobalFullControlCred: root:sundance

Licenses:

- core:

- non: HHLVBTQFEFPBBG

- add: BIOVBTQFEFPBBG

- dataprotection:

- non: JYMDITQFEFPBBG

- add: XNOICYBAZOOBBG

- provisioning:

- non: VSKSITQFEFPBBG

- add: JIMXCYBAZOOBBG

- disasterrecovery:

- non: THGWJTQFEFPBBG

- add: HXHBEYBAZOOBBG

LoginCred:

- testu1:testp1

- ctbuild:11ctbuild

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: mgt-ln07.lab.netapp.com

OS: linux

Software:

- HostAgent:

ManageCred: admin:private

MonitorCred: guest:public

OS: solaris

Plugins:

- SRM

- SAN

Port: 4092

ReleasePath: /rlse/Rcaffeine2.3.2xN/release

big\_path: /u/ctbuild/srmC-bigpath

small\_path: /u/ctbuild/srmC-bigpath/h

- HostAgent: {}

OSSV:

NDMPCred: test:1testtest

SVPirHost: mgt-ln07.lab.netapp.com

SVPriDirs:

- mgt-ln07:/ossvC-01

- mgt-ln07:/ossvC-02

- mgt-ln07:/ossvC-03

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: mgt-ln08.lab.netapp.com

OS: linux

Software:

- DFM:

GlobalFullControlCred: root:sundance

Licenses:

- core:

- non: HHLVBTQFEFPBBG

- add: BIOVBTQFEFPBBG

- dataprotection:

- non: JYMDITQFEFPBBG

- add: XNOICYBAZOOBBG

- provisioning:

- non: VSKSITQFEFPBBG

- add: JIMXCYBAZOOBBG

- disasterrecovery:

- non: THGWJTQFEFPBBG

- add: HXHBEYBAZOOBBG

LoginCred:

- testu1:testp1

- ctbuild:11ctbuild

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: mgt-ln13.lab.netapp.com

OS: linux

Software:

- DFM:

GlobalFullControlCred: root:sundance

Licenses:

- core:

- non: HHLVBTQFEFPBBG

- add: BIOVBTQFEFPBBG

- dataprotection:

- non: JYMDITQFEFPBBG

- add: XNOICYBAZOOBBG

- provisioning:

- non: VSKSITQFEFPBBG

- add: JIMXCYBAZOOBBG

- disasterrecovery:

- non: THGWJTQFEFPBBG

- add: HXHBEYBAZOOBBG

LoginCred:

- testu1:testp1

- ctbuild:11ctbuild

SystemLoginCred: root:sundance

- !perl/harness::HostSystem

Name: sig

OS: windows

Software:

- HostAgent:

CIFSCred: Administrator:sundance

ManageCred: admin:private

MonitorCred: guest:public

OS: windows

Plugins:

- SRM

- SAN

Port: 4092

ReleasePath: /rlse/caffeine/Rcaffeine2.5xN/release

big\_path: C:\srmC-bigpath

small\_path: C:\srmC-bigpath\stuff

- OSSV:

NDMPCred: test:1testtest

SVPirHost: sig

SVPriDirs:

- sig:C:\ossvC-01

- sig:C:\ossvC-02

- sig:C:\ossvC-03

SystemLoginCred: Administrator:tryme

--- Filers

---

- !perl/harness::Filer

AdminCred: ctbuild:11ctbuild

LoginCred: root:tryme

NDMPCred: root:tryme

Name: mgt-reg-u10.rtp.netapp.com

RLMHost: mgt-reg-u10-rlm

RLMIP: 10.60.194.42

SMLicense: UUNLDQC

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-reg-u10.rtp.netapp.com

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: pFiler

- !perl/harness::Filer

ASISLicense: OSFYRVH

LUNPaths:

- mgt-u71:/vol0/lun\_C1

- mgt-u71:/vol0/qt4luns/lun\_C2

- mgt-u71:/vol0/lun\_C3

LoginCred: root:tryme

NDMPCred: root:tryme

Name: mgt-u71.lab.netapp.com

SMLicense: UUNLDQC

SVSecHost: mgt-u71.lab.netapp.com

SVSecLicense: KVPYRVH

SVSecVols:

- mgt-u71:/ant\_sv\_2nd\_C1

- mgt-u71:/ant\_sv\_2nd\_C2

ScratchFlexVols:

- /scr\_fv\_C\_01

- /scr\_fv\_C\_02

- /scr\_fv\_C\_03

- /scr\_fv\_C\_04

ScratchNonLAs:

- ANT\_C1

- ANT\_C2

ScratchQtrees:

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01a

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01c

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01d

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01e

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01f

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01g

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01h

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01i

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01j

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02a

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02b

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02c

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02d

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02e

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02f

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02g

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02h

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02i

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02j

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03a

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03b

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03c

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03d

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03e

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03f

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03g

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03h

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03i

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03j

ScratchTradVols:

- /ANT\_TRADVOL

SyncSMLicense: IEOAOTB

TestModProp:

DRMMirror: host2

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-u71.lab.netapp.com

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: pFiler

- !perl/harness::Filer

ASISLicense: OSFYRVH

LoginCred: root:tryme

NDMPCred: root:tryme

Name: mgt-u70.lab.netapp.com

SMLicense: UUNLDQC

SVPriDirs:

- mgt-u70:/ant\_sv\_pri\_C1/qtr\_01

- mgt-u70:/ant\_sv\_pri\_C1/qtr\_02

SVPriHost: mgt-u70.lab.netapp.com

SVPriLicense: YAYCNCG

ScratchFlexVols:

- /scr\_fv\_C\_01

- /scr\_fv\_C\_02

- /scr\_fv\_C\_03

- /scr\_fv\_C\_04

ScratchNonLAs:

- ANT\_C1

- ANT\_C2

ScratchQtrees:

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01a

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01c

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01d

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01e

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01f

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01g

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01h

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01i

- /scr\_qtr\_C\_01/scr\_qtr\_C\_01j

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02a

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02b

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02c

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02d

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02e

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02f

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02g

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02h

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02i

- /scr\_qtr\_C\_02/scr\_qtr\_C\_02j

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03a

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03b

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03c

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03d

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03e

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03f

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03g

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03h

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03i

- /scr\_qtr\_C\_03/scr\_qtr\_C\_03j

ScratchTradVols:

- /ANT\_TRADVOL

SyncSMLicense: IEOAOTB

TestModProp:

DRMMirror: host1

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-u70.lab.netapp.com

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: pFiler

--- NISInfo

---

- Servers: 172.19.63.129

User: ctbuild

--- vFilers

---

- !perl/harness::Filer

HostingFilerName: mgt-u70

LoginCred: root:tryme

Name: mgt-vf195

ScratchFlexVols:

- /vf195\_scr\_fv\_C\_01

- /vf195\_scr\_fv\_C\_02

ScratchQtrees:

- /vf195\_scr\_qtr\_C\_01/scr\_qtr\_C\_01a

- /vf195\_scr\_qtr\_C\_01/scr\_qtr\_C\_01b

- /vf195\_scr\_qtr\_C\_01/scr\_qtr\_C\_01c

- /vf195\_scr\_qtr\_C\_01/scr\_qtr\_C\_01d

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-vf195

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: vFiler

- !perl/harness::Filer

HostingFilerName: mgt-u70

LoginCred: root:tryme

Name: mgt-vf196

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-vf196

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: vFiler

- !perl/harness::Filer

HostingFilerName: mgt-u70

LoginCred: root:tryme

Name: mgt-vf197

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-vf197

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: vFiler

- !perl/harness::Filer

HostingFilerName: mgt-u71

LoginCred: root:tryme

Name: mgt-vf198

ScratchFlexVols:

- /vf198\_scr\_fv\_C\_01

- /vf198\_scr\_fv\_C\_02

ScratchQtrees:

- /vf198\_scr\_qtr\_C\_01/scr\_qtr\_C\_01a

- /vf198\_scr\_qtr\_C\_01/scr\_qtr\_C\_01b

- /vf198\_scr\_qtr\_C\_01/scr\_qtr\_C\_01c

- /vf198\_scr\_qtr\_C\_01/scr\_qtr\_C\_01d

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-vf198

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: vFiler

- !perl/harness::Filer

HostingFilerName: mgt-u71

LoginCred: root:tryme

Name: mgt-vf199

ZAPI: !perl/Zapi::ZDapi

api\_name: ''

api\_param: ~

last\_output: ~

server: !perl/NaServer

debug\_style: ''

major\_version: 1

minor\_version: 0

password: tryme

port: 80

server: mgt-vf199

server\_type: ''

servertype: FILER

style: LOGIN

transport\_type: HTTP

url: /servlets/netapp.servlets.admin.XMLrequest\_filer

user: root

vfiler: ''

xml: ''

filerType: vFiler

--- FTPHost

---

- Name: vineland.eng.netapp.com

Resource Data

============ENVIRONMENT DATA============

---

DFM TEST ENVIRONMENT:

Host: mgt-ln13.lab.netapp.com

Kernal Release: 2.6.9-42.ELsmp

Kernal Version: '#1 SMP Wed Jul 12 23:32:02 EDT 2006'

Machine: x86\_64

Operating System: Linux

Platform: x86\_64

Processr: x86\_64

RESOURCE TEST ENVIRONMENT:

- mgt-reg-u10.rtp.netapp.com:

hostOSVersion: 7.3.1

hostType: Filer

productFirmwareVersion: CFE 3.1.0

productModel: FAS3050

- mgt-u71.lab.netapp.com:

hostOSVersion: 7.3.1.1

hostType: Clustered Filer

productFirmwareVersion: 3.0

productModel: FAS2020

- mgt-u70.lab.netapp.com:

hostOSVersion: 7.3.1.1

hostType: Clustered Filer

productFirmwareVersion: 3.0

productModel: FAS2020

Revision History

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_