DFM: Build Guide

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

1. Overview

1.1 Why did we do it?

1.2 How did we do it?

1.3 What's New

2. Howtos

2.1 How to build DFM

2.1.1 Tips for Windows developers

2.1.2 Common Problems for VisStudio Dev

2.1.3 Using a VM in RTP for VisualStudio development

2.2 How to build specific targets

2.3 How to clean the build

2.4 How to add new source files to the build

2.5 How to add a new executable to the build

2.6 How to add a new library to the build

2.7 How to modify NDMP messages

2.8 How to generate files during a build

2.9 How to generate files for cscope, tags, etc.

3. Common Build Problems

3.1 sync\_shared acting up

3.2 Can't start new winbuild jobs

3.3 Your shell and rsh

3.4 Hung Windows process

4. CMake

4.1 CMake mini style guide

4.1.1 Indentation

4.1.2 Naming Conventions

4.1.3 List Formatting

4.2 Learning CMake

4.2.1 Commonly used CMake constructs

4.2.2 Useful CMake script examples

4.2.3 Getting CMake To Generate Win64 Build Targets in Champagne.sln on the Desktop

5. Build Policies

5.1 No generated files allowed in src

5.2 Generated headers must all be made first

5.3 Single header file namespace

6. Internals

6.1 CMake Internals

6.2 GCC 4

6.2.1 GCC 4 compiler flags

6.3 distcc

6.4 Visual Studio 2005

6.5 winbuild

6.6 IncrediBuild

6.7 Unix Packaging

6.8 Windows Packaging

7. Miscellaneous Build System Changes

7.1 libproduct vendor library

7.2 Client bundling

7.3 No ctod

7.4 Relocation of 3rd party software

7.5 Optimization and debug symbols

8. Build Time Code Generation

8.1 libsoapd - Web Service for ZAPIs

9. What's Next

9.1 Library re-organization

9.2 Convert to dynamic libraries

9.3 Remove unused #includes

10. Useful References

1. Overview

DFM switched over to a new distributed build system in June 2008. This document is the central repository for all build related information since the switchover. All information related to the new build system must go here. IMPORTANT: All other documents including DFM Hitchhiker's Guide should not duplicate build related information and should instead link back to this document.

1.1 Why did we do it?

Why did we need a build overhaul? Some numbers:

• Nightly build took 15+ hours

• Linux full compile took ~1 hour

• Solaris full compile took ~2 hours

• Windows full compile took ~3 hours

• Single line change took 5 minutes minimum to rebuild each binary.

In short, it was driving everyone crazy.

1.2 How did we do it?

How did we do it? First, some numbers:

• Nightly build without Windows packaging (not ready yet) takes 50 minutes

• Linux full compile takes 8 minutes

• Solaris full compile takes 8 minutes

• Windows full compile takes 12 minutes

• Single line change takes a minute to rebuild all binaries.

The following is just an overview. Read the Internals section for more details.

We have changed many parts of the build process. Migration to CMake is an important part of this work. CMake is a cross-platform make and vcproj generator. Using CMake we maintain one set of build scripts for all platforms. From these scripts CMake generates native build scripts with accurate inter-library dependencies. Accurate depedencies allow DFM to build reliably in distributed environments.

Both Unix and Windows DFM builds are distributed . For Unix we use an open source package called distcc and leverage the already existing engsup-maintained x64-build farms. Kevin Nomura, our in-house GCC wizard, built us two GCC 4 cross compilers which allow us to build the linux x86 and solaris sparc platforms on those fast machines.

On Windows we use IncrediBuild to distribute compile workloads among the brand-new quad-core machines in SVL and NB. IncrediBuild has been in use for a while but had to be turned off for nightly builds because of reliablity issues. It was difficult to maintain the old vcproj files with a high degree of confidence and CMake gives us that boost to use IncrediBuild reliably. We have also upgraded to Visual Studio 2005.

We further reduced the nightly build time by cutting the total number of builds by half. We used to build each set of binaries twice - one set for NetApp and the other for IBM. Many vendor-specific constants were compiled into the code, so we had to do it that way. In the new build, we isolated all vendor-specific information into a dynamic library and modified the code to use the interface provided by that library to retrieve vendor information at run time. Now NetApp and IBM happen as one build. The packaging steps install the correct vendor library.

1.3 What's New

There are many new things in the new distributed build system. The noteworthy items are:

• Migration to CMake 2.6.0.

• GCC 4 cross compilers

• Visual Studio 2005 Professional Edition

• Distributed compile using distcc on Unix and IncrediBuild on Windows.

• Cross compile Unix versions on x64 build farms.

• Build Windows version on new CfgEE winbuild farms.

• Single header file namespace.

• "Out-of-source" build. No generated files allowed in the 'src' directory

• Build once for both NetApp and IBM.

• Windows release binaries built with full symbols to match what we have been doing on Unix.

• Unix binaries ship with optimization (-O2) to match what we have been doing on Windows.

• Treat warning as error on Windows just like what we have been doing on Unix.

• All Unix binaries are stripped of symbols in shipping packages.

• Apache, openssh and Putty moved into champagne\_shared and brought into DFM as prebuilt binaries.

• Client install packages no longer included in developer's build.

• No more ctod. Must write our own forward declarations.

2. Howtos

2.1 How to build DFM

First, use getstree to get a p4 client workspace if you haven't already done so.

Caution: Do not use an existing pre-cmake client workspace. There are generated files and old winbuild configurations that could interfere with the new build.

Second, cd to the top of the client workspace. Run the script:

% ./make\_dfm

This script forks three processes to build DFM for all three platforms at the same time. The Unix jobs are sent using a script called '/usr/software/rats/bin/rehosts.pl' to one of the x64 linux build hosts. The Windows jobs are sent using '/usr/local/bin/winbuild' to one of the CfgEE winbuild machines. Note: The client workspace must be getstree-created for the winbuild machines to find it.

The build progress of these processes is gathered by 'make\_dfm' and sent to STDOUT. A build log is kept for each platform in 'build/{platform}/make\_dfm.log', where {platform} is either 'linux', 'solaris', or 'winbuild'.

The build binaries are kept in the 'build/{platform}/bin' directory. There are no sbin directories at this stage. (There is a separate staging step in packaging that stages the files in a directory structure that mirrors an actual installation. The sbin directory appears there).

In fact, as a policy, all files generated by the build go to the build directories. No generated files are allowed to be saved in the src directory.

'make\_dfm' has many useful options. For details, run:

./make\_dfm -h

Here are some useful examples:

• Before checking in your code, run 'make\_dfm' with no arguments. It builds the release configuration of all three platforms like a nightly build would.

• % ./make\_dfm

• To make a debug build, use the --debug. For example, this will make a linux debug build:

• % ./make\_dfm --debug --linux

• To make an installable package, use the --package option:

• % ./make\_dfm --linux --package

Note: The Windows package is not yet ready. The --package option only does staging on Windows.

2.1.1 Tips for Windows developers

Windows developers may use a virtual machine that is mostly setup for windows builds and just needs a little bit of effort to get going or they may choose to setup their own PC to do windows builds using Visual Studio development. The winbuild farm will always make windows builds but that linux environment is not suitable for working with VisualStudio and is only used to form installer packages as part of our release. If you prefer to build Windows DFM on your desktop instead of the winbuild farm, you need to install the software shown below and you may wish to read the section that follows that points out some common problems with environments that may lead to a broken build.

• Visual Studio 2005 Professional Edition

• CMake 2.6.0

Create a DFM client workspace. This workspace can be on your local hard disk or on a filer. Reusing the workspace you have already created for your Unix development is probably the most effortless. Hint: don't forget to run "perl sync\_shared -f" on your workspace if it has not been done before, or is out-of-date. This is likely the case if you are NOT reusing a your Unix workspace. The "-f" option may not be necessary, but sometimes components are left out.

Next, pick a directory where you want the object files and binaries to go. This can be on a local disk or on a filer. For performance reasons, a local disk is preferred.

Caution: If you pick a CIFS directory that is mounted via the \ead.netapp.com redirector and you are running Windows XP, you might experience intermittent delayed-write failures when the compiler writes large object files to it. For this reason we do not recommend putting the build directories on a CIFS-mounted filer. Putting them on an NFS-mounted filer is acceptable.

Follow these steps to build Windows DFM locally. Assuming your workspace is s:\buildw1\scratch\foo\p4\aw\main, and you have picked c:\foo\main\build\win as the directory for the object files:

a. mkdir c:\foo\main\build\win

b.

c. c:

d. cd c:\foo\main\build\win

e.

f. cmake s:\buildw1\scratch\foo\p4\aw\main\src

g.

h. As a result of the above step, CMake has created Champagne.sln and all the vcproj files. Open Champagne.sln in Visual Studio. Build the ALL\_BUILD project. Caution: Cmake should build project files compatible with Visual Studio 2005. If VS warns that the project files are the wrong version, then delete your win directory and start again from step a. In step c., add the option: -G "Visual Studio 8 2005"

Alternatively, you can build it on the command line. Open a Visual Studio 2005 Command Prompt, and run:

C:\> devenv.com Champagne.sln /build /config "RelWithDebInfo|Win32"

Use "Debug|Win32" if you want a debug build.

Note: devenv.com sends its build output to STDOUT. devenv.exe is silent, but you can save the output to a file using the /out option.

The binaries will be saved in c:\foo\main\build\win in this example.

Remember to setup VisualStudio 2005 Working Folder If you are going to debug in Visual Studio 2005 recall that you must select 'Project->properties->Debugging' and then set working dir to C:/Program Files/NetApp/DataFabric/DFM/bin just like visual studio 2003.

There are additional setup steps when no windows installer exists.

As of June 2008 the new make process does not form windows installers. However, if you have to try the Windows version before the installer becomes ready, there is a way to do it.

In each nightly, we stage all the Windows DFM files in the same directory structure as it appears in an installation. These files are in /build/winbuild/staging. You can follow these manual procedures to install it:

a. If you don't already have a working DFM, install a pre-cmake version. The last pre-cmake version is RchampagneN\_080529\_2300.

b. Run 'dfm service stop'. Verify that all services have stopped.

c. Locate the new nightly you want to install. Copy the three directories (DFM, Perl, Sybase) from the nightly's build/winbuild/ staging directory into C:\Program Files\NetApp\DataFabric, replacing the three subdirectories by the same names.

d. Also from the new code tree src folder copy all files from src/xerces/xerces-c\_2\_8\_0/nt/lib to target machine DataFabric/DFM/bin folder for xml libs.

e. Run 'dfm service start sql'

f. Run 'dfm service upgrade'

g. Run 'dfm service start'

Special windows perforce issue for Mac format files Visual Studio 2005 does not like lines in files that end in non-standard end of line sequences. To prevent problems one must setup the P4 client to pull files using 'unix' files which is in the P4Win tool as a pulldown menu. Also in the works is a fix to sync\_shared where we will add a command line option for the shared components to fix this issue.

2.1.2 Common Problems for VisStudio Dev

Using a version of cmake other than 2.6-patch0 can lead to a failed make which generally leads to errors about include file apierrorno.h. Check version using 'cmake --version'. Note that you MUST run cmake fresh again to really solve your problem after this.

The Incorrect java version or more likely a missing JDK1.6 environment variable which is the path to the jave install. This generally leads to errors in soap code generation. We know that jre1.6.0\_03 works. Any other rev you are on your own. For that env var JDK1.6 = C:\Program Files\Java\jre1.6.0\_03 Note that you MUST run cmake fresh again to really solve your problem after this.

When the incorrect version of rpcgen.exe is picked up errors happen when we process the snapmirror and other .x files. If you see snapmirror\_xdr.c errors this is likely. The most common cause is after installing cygwin into cygwin/bin there will be an incompatible rpcgen.exe in the bin folder. Rename it to a bad name so it is not found OR it also works, maybe even better, to have a full champagne tree pulled and then replace cygwin/bin/rpcgen.exe with src\rpc\nt\bin\Rpcgen.exe. Note that you MUST run cmake fresh again to really solve your problem after this.

2.1.3 Using a VM in RTP for VisualStudio development

Windows developers may use a virtual machine that has been setup with a windows 2003 OS as well as VisualStudio and a host of other packages generally used for windows development.

If you have any problems be sure you have done the steps to the default VM image described in the section on common problems with environments that may lead to a broken build.

Windows dev requires knowing about p4win and manually running cmake and some other things that may be covered in the earlier section on windows development tips.

Be sure to first Reserve virtual build machine We reserve a virtual dev machine in RTP by editing the Dev VM Reservation Wiki

(We will try to get these on the lab scheduler but for now use above Wiki for these special dev VMs)

Get setup as a local Admin on your VM Get local administrator ability on your VM for your netapp login. Contact enghelp+rtp@netapp.com to have them set this up for your login

Remote Desktop To Your VM We login to the Virtual Dev Build machines using our NetApp single signon in NETAPP domain.

First Time You Start VisualStudio For the first time you start VisualStudio you will need to answer a few questions.

• select Visual C++ Development Settings.

• Go into Build->Configuration Manager and select RelWithDebInfo as the build. As of 4/2010 our 'Debug' build has 'issues' with debug libraries. How to do a clean Build If you are not able to select the ALL\_BUILD project as active project and then do a Clean and the Build of this project there is something wrong with your environment or the build.

Adjustments To The Default VM Image required once

Hopefully this section will disappear or grow smaller as we generate a new VM image so new people will be on the air even quicker. As of April 2010 we had to do these things.

As of early 2010 the VM image was not quite as perfect as it needs to be so some additional steps are required to get it working for builds. These steps will be described below.

We must be sure cygwin/bin/rpcgen.exe is NOT used by cmake. After you have pulled a full champagne tree, including perl sync\_shared, then replace cygwin/bin/rpcgen.exe with src\rpc\nt\bin\Rpcgen.exe. This is required because we must not allow tools in C:\cygwin\bin from the default system PATH variable to be found or parts of the make will fail. We all like cygwin so it is best to rename cygwin/bin/rpcgen.exe OR replace as described.

The make which will be done under your user Single Sign On login requires a Local Settings/Temp folder and that may or may not be generated automatically so maybe maybe make one like this (with your user name) C:\Documents and Settings\yourLoginName\Local Settings\Temp (I am not entirely sure if this step is required but it is easy)

We think that the installed java does not have the sdk and we may need it. You need to get and install jre-6u3\_1\_6\_0\_03-windows-i586-p.exe and I am not sure if any other rev works so feel free to experiment. Failure would be in the cmake step. If you cannot find it anywhere, there is on in http://web.netapp.com/~mjstn/

Check and modify as required these windows environment variables that can be found from My Computer right click -> properties -> Advanced tab -> Environment Variables Create an environment var called P4PORT and set it to p4netapp:1666 (Note that in rtp this will resolve to .rtp.netapp.com local perforce) Create an environment variable called JDK1.6 and set it to C:\Program Files\Java\jre1.6.0\_03 (use path to your jre if you did not install 1.6.0\_03 just now)

If you use p4win for perforce then you will need to put p4netapp:1666 in Settings -> Options in the P4 port box in Server Settings.

2.2 How to build specific targets

During development, it is often useful to shorten the code-compile-test cycle by building only the targets being tested. You can do that with 'make\_dfm'. For example, run this to build just 'dfmmonitor':

% make\_dfm --linux monitor

To see a list of possible targets, run:

% make\_dfm --linux help

Note: This 'make\_dfm' feature is not yet available for winbuild.

2.3 How to clean the build

All generated files go under the build directories. No generated files are in src. To clean, simply remove the entire build directory. For example, to clean linux,

% rm -rf build/linux

As an alternative, run:

% ./make\_dfm -fff --linux

This removes 'build/linux' and run 'sync\_shared -f' before starting the build.

2.4 How to add new source files to the build

This is simple.

a. Locate the directory that holds the library or executable to which you want to add new source files.

b. Open up 'CMakeLists.txt' in your favorite editor in that directory.

c. Add any new header files to the SUBDIR\_HDRS list and any new source files to the SUBDIR\_SRCS list. One item per line is required. We want to keep our gatekeepers happy and make 'p4 resolve' easy for them.

2.5 How to add a new executable to the build

Use this as an example:

src/monitor/CMakeLists.txt

2.6 How to add a new library to the build

Use this as an example:

src/libmon/CMakeLists.txt

2.7 How to modify NDMP messages

From time to time we find it necessary to modify NDMP messages by changing files such as libndmp/snapvault.x. Normally the make would then use rpcgen to genarate two files like snapvault.h and snapvault\_xdr.c for both the linux and solaris makes. This does not function so here is a way to generate the needed files and submit them to make NDMP new messages available.

In this example we assume a change is done to snapvault.x

1) In the zephyr client in src/libndmpd use p4 edit to open

snapvault.x

src.linux/snapvault.h, src.linux/snapvault\_xdr.c

src.solaris/snapvault.h, src.solaris/snapvault\_xdr.c

2) Make changes to snapvault.x as required.

3) Get on a linux make machine in folder libndmp/src.linux then run:

/usr/software/rats/bin/rehost2.pl --hostpat="LIN Platform1x" make -f Makefile.rpc

4) Get on a unix make machine in the folder libndmp/src.solaris then run:

/usr/software/rats/bin/rehost2.pl --hostpat="SunOS" make -f Makefile.rpc

You will now have 5 opened files in your zephyr client including the snapvault.x,

and the updated \_xdr.c and .h files for in this case snapvault.

Debug and then get the changes reviewed.

5) Use 'p4 submit' and the files will go into zephyr.

The next time a shared component for zephyr/src/libndmp is sync\_shared into a DFM client the 5 files will be there ready for the dfm build. Sometimes we use a different zephyr like how we used zephyr-isis for champagne-isis in late 2008. So the zephyr client would of course be a client to a zephyr-isis in that case.

2.8 How to generate files during a build

First of all, any generated header files that might be included by other libraries must be generated before any source files are compiled.

Next, learn about CMake functions add\_custom\_command and add\_custom\_target .

For a simple example, see:

NETAPP\_GENERATE\_HH in src/CMake/NetApp.cmake

For an advanced level course, see:

src/libdfmapi/GenerateShared.cmake

2.9 How to generate files for cscope, tags, etc.

Run 'make\_dfm' with the appropriate options:

--ctags

Create a tags file to be used by vi.

The tags file will be generated in the Linux build farm.

--etags

Create a TAGS file to be used by emacs.

The TAGS file will be generated locally.

--cscope

Create a cscope database for browsing source code.

The cscope database will be generated locally because the Linux-built

cscope datasbase does not work on Solaris.

3. Common Build Problems

3.1 sync\_shared acting up

This happens intermittently to new getstree-created workspaces. The getstree script invokes sync\_shared at the end. For some unknown reason, sync\_shared sometimes fails to pull in all the files from the shared components. It gives no warnings when that happens. The first sign of trouble usually appears during build where CMake errors out with messages similar in nature to this one:

CMake Error at libndmp/GenerateShared.cmake:11 (message):

Could not find rpcgen

Call Stack (most recent call first):

CMakeLists.txt:249 (include)

-- Configuring done

Tue Jun 3 23:39:05 2008: \*\*\* Error running CMake

The missing files are always in the shared components.

Solution: Run 'sync\_shared -f'.

3.2 Can't start new winbuild jobs

If you see a make\_dfm error like this:

win: 19:39:44 Run Script Makeaw.pl

win: 19:39:44 Job JCL File slam-Rchampagne.jcl

win: 19:39:44 Job Log File slam-Rchampagne.log

win: 19:39:44 Script Log File winbuild.log

win: 19:39:44 Submitting Windows Job slam-Rchampagne...

win: 19:39:44 \*\*\* Cannot Submit Windows Job slam-Rchampagne, \

slam-Rchampagne.Waiting file exists

You tried to kill a winbuild job before it had finished by killing the 'make\_dfm' or 'winbuild' process. This only kills the unix side of the job. It does not stop the build from proceeding. Once a winbuild job has started, there is no way to kill it without submitting an engsup ticket.

Solution: Wait until the last job has finished by verifying that there is no [job-name].Running file. Then remove the [job-name].Waiting file before starting a new job.

3.3 Your shell and rsh

distcc depends on rsh to ship compile jobs over the network to other build hosts. Make sure you .bashrc, .cshrc, etc. are clean. An unclean example is this:

% cat .cshrc:

bash

In this case the default shell was /bin/csh, but the user actually wanted to use bash. This will cause mysterious build errors like this:

bash: line 1: DIST00000001ARGC0000000bARGV00000051/usr/software/build/\

compilers/i386-redhat8-linux-gcc-4.2.3/bin/i386-redhat8-linux-gcc\

ARGV00000011-Wno-pointer-signARGV00000003-O2ARGV00000002-gARGV00000014\

-fno-strict-aliasingARGV00000005-WallARGV00000007-WerrorARGV00000002\

-oARGV00000025CMakeFiles/libnetapplm.dir/lickey.c.oARGV00000002\

-cARGV00000045/x/eng/buildw1/scratch/foo/p4/aw/main/src/libnetapplm/lickey.c\

DOTI00023428#: No such file or directory

Solution: Use this web page to change your default shell:

http://topaz.corp.netapp.com/useradmin/

3.4 Hung Windows process

Occasionally, a winbuild job hangs. If you do not see any new log entries written to 'build/winbuild/build.log', the build process on the winbuild machine can potentially be hung.

Solution: Submit an engsup ticket to have the job killed.

4. CMake

CMake is an open source, cross platform 'make' generator. It reads platform- and compiler-independent build scripts and generates platform-native make and vcproj files.

The major values of CMake to DFM are as follows:

• CMake frees us from maintaining two sets of build scripts. Maintaining Unix make files by hand was manageable, but we were never able to keep accurate inter-library dependencies. Editing Windows vcproj files was a nightmare. It was quite a challenge for simple tasks. It was extremely difficult and error prone when the tasks involved any pre- or post-build steps. CMake, even with its slightly awkward syntax, makes our lives much easier.

• CMake does a great job of maintaining proper inter-project dependencies. This is vital to our ability to reliably compile our code in a highly distributed manner. Before CMake, running 'make -j' on our hand-crafted set of make files was never reliable.

• Upgrading Visual Studio in the future will be much easier.

4.1 CMake mini style guide

CMake has no offical style guide. We came up with our own based on the keep-our-gatekeepers-happy principle. When faced with a choice think about your gatekeepers and choose a style that makes it easier for them to resolve your p4 conflicts.

Here is the short list:

4.1.1 Indentation

Follow the NetApp Perl Coding Standard on indentation.

• Indent with spaces, not tabs, nor a mix of tabs and spaces.

• Use four-column indentation levels - indentation should be 4 spaces per level of indentation.

• If you are a vim user, add this to your .vimrc:

• autocmd FileType cmake set tabstop=4 expandtab shiftwidth=4

Optionally, you can set this to catch non-conforming tabs in CMake scripts:

autocmd FileType cmake set list listchars=tab:>trail:-

4.1.2 Naming Conventions

• CMake functions and macros should be invoked in all lowercase and be defined in all uppercase.

• Use underscores to separate words in multiword identifiers.

• Variables should be used and defined in all uppercase.

4.1.3 List Formatting

• A long list should be separated by newlines, with one item on each line, and the closing parenthesis on its own line. For example,

• set(LONG\_LIST

• item\_a

• item\_b

• item\_c

• )

• If a list starts off small, but has the potential of growing in size, follow the above guidelines.

4.2 Learning CMake

The best way to learn CMake is by reading the Online Man Page on some of the more important constructs, and by reading existing CMake scripts.

4.2.1 Commonly used CMake constructs

Refer to the Online Man Page of CMake. Useful constructs are:

• Variables are defined with set .

• Read add\_library to learn how to add a library (both static and dynamic) to CMake. However, for DFM we should use the wrapper functions netapp\_add\_liband netapp\_add\_dll .

• Read add\_executable to learn how to add an executable to CMake. However, for DFM we should use the function netapp\_add\_exe instead.

• target\_link\_libraries links a target to other libraries. This is a very powerful mechanism to ensure correct inter-library depedencies. This is a must read.

• configure\_file is used extensively to convert input templates into source files. It is also used to generate Windows resource file with the correct vendor and build info.

• Read add\_custom\_command and add\_custom\_target to learn about ways to generate build input files with scripts.

Note:Read this build policy first before creating CMake rules for your first generated files.

• For a simple example, look at the NETAPP\_GENERATE\_HH macro in src/CMake/NetApp.cmake.

• For a more complicated example involving zapidoc, look at src/libdfmapi/GenerateShared.cmake.

• For an example of how add\_custom\_target is used, see src/man/CMakeLists.txt.

• Some useful variables are:

• CMAKE\_BINARY\_DIR

• CMAKE\_SOURCE\_DIR

• CMAKE\_CURRENT\_BINARY\_DIR

• CMAKE\_CURRENT\_SOURCE\_DIR

4.2.2 Useful CMake script examples

The CMake scripts are all under the srcdirectory.

• An example of a CMake script for a simple library:

• src/libmon/CMakeLists.txt

• An example of a simple executable:

• src/monitor/CMakeLists.txt

• To see examples of CMake macros and functions, read:

• src/CMake/NetApp.cmake

4.2.3 Getting CMake To Generate Win64 Build Targets in Champagne.sln on the Desktop

When running CMake, on the command line on your 64 bit Windows desktop or VM it will not automatically generate Win64 build targets. To get those targets, you will need to pass in -G "Visual Studio 8 2005 Win64i" -DCPU\_ARCH=x64 which will give you a CMake invocation that looks like this:

path-to-build-output-dir> cmake <path-to-your-source-directory> -G "Visual Studio 8 2005 Win64i" -DCPU\_ARCH=x64

5. Build Policies

To maintain maximum build throughput and stability we have created some new coding policies.

5.1 No generated files allowed in src

To make sure all three platforms can be built at the same time independently of each other, there is now a strict policy that no generated files be allowed to be saved in the src directory. All output files must go to the build directories.

To view this policy from an alternate angle, all generated files must be written to platform dependent locations. For example, file 'foo' generated by windows and linux must no longer be saved in the same directory 'bar'. Instead, they must go into 'bar/windows' and 'bar/linux', respectively. For example, the old test2/lib/Zapi/ZDapiErrno.pm was a generated file that would have violated this policy because all three platform build this file at that same location.

CMake makes it very easy to refer to the build output directories in the CMake scripts. Use the variables CMAKE\_BINARY\_DIR and CMAKE\_CURRENT\_BINARY\_DIR .

5.2 Generated headers must all be made first

To maintain cross platform compatibility, all generated header files must be made at the beginning of the build before any source files are compiled. We created this build policy to work around a shortcoming of Visual Studio 2005. If a generated file is built in one project and included in a source file in another project, Visual Studio does not always wait for the generated header to be made first before compiling the source file in the other project. This can potentially produce incorrect build. To completely eliminate this potential headache, we require that all generated headers be made at the beginning of the build. CMake makes this very easy. Just append the target for the generated headers to the CMake variable DFM\_GENERATED\_TARGETS.

5.3 Single header file namespace

You must #include a header file only with its file name. No directories are allowed in the path. For example, the following is okay:

#include "obj.h"

But this is not:

#include "libcmd/obj.h"

The above code in fact will no longer compile. The src directory has been removed from the include search paths on all three platforms, and all #includes in the form above have been corrected.

We created this policy to discourage developers from blindly using a library for which it is not designed to be used. If a library is not already in the include search paths, using that library will cause the compile to fail. This should give the developer a signal to check the following possibilities:

• the library needs to be added to the include search path and linked in

• a portion of the library needs to be moved to another library

• or the library should not be used at all.

One result of this policy is that we effectively have a single header file namespace for all of the DFM code base. A very short header file name might collide with those that already exist in another library.

However, this is not necessarily a bad thing. By uniformly referring all #includes by filename only, it makes it easier to reorganize our libraries from the current tangled mess into a more sensible structure. This is the first step in the right direction.

Also, having a single namespace helps Intellisense in Visual Studio find the correct files. In the past, for example, Intellisense sometimes mistakenly identified constructs in libdfmcore/Dataset.hpp as in libcmd/Dataset.hpp. We renamed libcmd/Dataset.hpp to libcmd/DatasetCli.hpp to correct the problem. Many files have also been similarily renamed.

Do not circumvent the removal of the src directory from the include search paths using '..' in the path. For example, this is not acceptable:

#include "../libcmd/obj.h"

6. Internals

6.1 CMake Internals

XXX - Talk about the toolchain files. Discuss where the object files go inside the build directory.

6.2 GCC 4

We use GCC 4.2.3 cross compilers to build linux x86 and solaris sparc versions of DFM on the x64 linux build farm. (The x64 build machines are identified by the string 'LIN Platform3x' in the file /x/eng/site/ALTHOSTS).

These compilers can be found here:

/usr/software/build/compilers/i386-redhat8-linux-gcc-4.2.3

/usr/software/build/compilers/sparc-sun-solaris2.8-gcc-4.2.3

The system header files used by the cross compilers can be found here:

/usr/software/build/compilers/i386-redhat8-linux-gcc-4.2.3/sysroot

/usr/software/build/compilers/sparc-sun-solaris2.8-gcc-4.2.3/sysroot

GCC 4 is much improved. To see what was changed since 3.3.3 which was the last version used, check out these GCC release notes:

• 4.2 release notes

• 4.1 release notes

• 4.0 release notes

• 3.4 release notes

Caution: Test build both Unix platforms before submitting your code. With -O2 the two GCCs can behave very differently. Your code can potentially fail on one but not the other.

6.2.1 GCC 4 compiler flags

The global GCC compiler flags should be set in:

src/CMake/CompilerFlags.cmake

Refer to that file for the most up-to-date set of options. If you add an option there, make sure it is well documented.

6.3 distcc

Our version of distcc can be found here (only visible on the x64 build hosts):

/usr/software/build/distcc/distcc-2.18.3/bin/distcc

distcc is a compiler front-end for distributing compile loads across several machines on a network. Instead of sending a compile job to gcc, CMake is configured to send it first to distcc where the arguments that are destined for gcc are parsed and the compile task separated into two distinct steps. The first step is performed locally (local here means the x64 build host that receives the 'make -j' request from the 'make\_dfm' script over rsh, not the compute server running 'make\_dfm'). It invokes gcc to preprocess the source file into a fully preprocessed source. The file is then shipped to a build machine over the network using a private distcc protocol where gcc is invoked to compile the preprocessed source into an object file. The object file is sent back over the network to the sending machine where it is saved to the file system.

To determine which machine receives a compile job, we use the althost mechanism maintained by the build team. CMake is configured to send each job to a distcc wrapper in src/util/distcc/distcc-gcc. The script retrieves a list of candidate build hosts from the althost "database" and passes it to distcc. distcc then sends the compile job randomly to one of the candidates.

The althost "database" is capable of returning a sorted list of hosts that match a certain host pattern specified in /x/eng/site/ALTHOSTS, with the most lightly loaded ones on top. The host pattern for the x64-linux build farm is 'LIN Platform3x'.

To ship a preprocessed source over the network, distcc normally requires a daemon called distccd to be installed and started at boot time. This requires getting engsup to sign off on maintaining those daemons. Rather than putting that burden on engsup, we employ a simpler alternative at the expense of a slightly higher startup cost.

distcc supports sending jobs over ssh. It logs on to a build host as the user, starts distccd there, and tunnels the distcc traffic back and forth. Because ssh has a high startup cost and the configuration of passwordless ssh session is error prone, we have decided to use rsh instead. rsh is very widely used at NetApp and is the protocol used by the ONTAP team to distribute their compile jobs.

In order for distcc to tunnel rsh traffic reliably, however, we have to patch the official distcc build to disable non-blocking I/O over the rsh/ssh protocol. Most rsh daemons do not handle non-blocking I/O reliably. The patch is in src/util/distcc/distcc-rsh-patch.

We applied two other patches to our version of distcc. The first one is here. It patched distcc to choose randomly from the list of candidates passed in through the DISTCC\_HOSTS environment variable instead of preferring the one at the head of the list. The althost 'database' is updated at a finite interval. By randomly choosing among the candidates we minimize the chance of overloading the one at the top of the list before althost is updated.

The second patch is documented here. It limits how many non-compile gcc jobs (like preprocessing and linking) can be performed on the local machine at the same time. By putting an upper bound on those jobs we can use the make '-j' option to start as many jobs in parallel as possible while allowing distcc to regulate the number of active jobs without overloading the local machine.

6.4 Visual Studio 2005

Visual Studio 2005 is now capable of utilizing all CPU cores when compiling.

Visual Studio issues many new warnings about the use of certain functions it deems unsafe. Warnings like this are very common:

C:\Program Files\Microsoft Visual Studio 8\VC\include\string.h(73) : see declaration of 'strcpy'

Message: 'This function or variable may be unsafe. Consider using strcpy\_s instead.

To disable deprecation, use \_CRT\_SECURE\_NO\_DEPRECATE. See online help for details.'

We have enabled these compiler flags in "src/CMake/CompilerFlags.cmake" to disable those warnings:

-D\_CRT\_SECURE\_NO\_DEPRECATE

-D\_CRT\_NONSTDC\_NO\_DEPRECATE

-D\_SCL\_SECURE\_NO\_DEPRECATE

6.5 winbuild

We use the build-team-maintained winbuild infrastructure for building DFM nightly builds and private developer test builds for those of us who choose not to build Windows DFM on the desktop. It consists of several groups of Windows servers, with servers within each group sharing a common set of software and comparable hardware configurations. The server group used by DFM is called cfgEE, which currently consists of five machines in SVL and five in NB (they are not ready as of June 5, 2008). Each server in the cfgEE group comes with all the software required to build DFM from scratch to an installable package.

The winbuild infrastructure is not trivial. How exactly a job is submitted differs between a private developer build and an official nightly build. However, the basic ideas are the same.

Let us use a developer build as an example. A winbuild job starts its life on the Unix side of the build when the user runs this script:

% make\_dfm --windows

The script forks a process and run '/usr/local/bin/winbuild'. The winbuild script gathers information like the location of the tree and the type of product being built and turns that information into a job control language (JCL) file. The file is saved to one of the job queues in /u/build/NTBF/Spool/. For DFM, the queue is /u/build/NTBF/Spool/CfgEE. Once the JCL file is saved to the queue, the winbuild script creates a [job-name].Waiting file at the top of the p4 client workspace and goes in a loop waiting for the job to finish.

Note: In NB, the job queues are in /u/btcwfarm/NTBF/Spool.

Nightly builds do not use '/usr/local/bin/winbuild'. The job queuing is done directly inside the '/usr/local/bin/fab' script which controls the entire nightly build process.

The windows side of the build starts when one of the Windows build server picks up the job from the queue. It creates a [job-name].Running file at the top of the client workspace and passes control to Makeaw.pl at the top of the champagne tree where the actual build takes place.

When the build is done, the Windows server removes the [job\_name].Running file and creates a [job\_name].Done file at the top of the tree. The winbuild script on the Unix side sees that change and exits.

Caution: Once a winbuild job has started, there is no way to kill it without submitting an engsup ticket. As a hack, you can cause the job to fail early by removing the 'build/winbuild' directory.

Note: There is no "althost" mechanism for the winbuild farm. A job in the queue is assigned randomly to a server on the farm.

Each p4 client workspace can only start one active job at a time. If you ctrl-C out of the winbuild process (or the 'make\_dfm' process that spawned it) before the job is done, you will not be able to start a new one until the current job has finished. Look for the existence of the [job-name].Running or the [job-done].Done files to tell the state of the job.

Important: If you kill the winbuild process after a job has been submitted but before it finishes running, you have to remove the [job-name].Waiting file after the job finishes or you will not be able to start a new winbuild job.

6.6 IncrediBuild

IncrediBuild distributes C/C++ compile jobs across machines in a local network. It is fully integrated with Visual Studio. It does not require any changes to source or project files. All it requires is the installation of an IncrediBuild agent.

There are two types of agents - a full agent initiates distributed jobs and participates in jobs initiated by other full agents as a remote computing resource. A helper agent only participates in jobs initiated by a full agent.

In our winbuild environment, there are 14 full agents in SVL. Five of these 14 agents are CfgEE machines. These machines can drive DFM build job. The rest of the machines, when they are free, participate as helper agents..

One limitation with IncrediBuild is that only one build job can be initiated per full agent at a time. There is an upper bound on how many DFM build jobs can be started on our CfgEE winbuild farm at any given time. If an agent is already busy when a new job is submitted, it will be blocked until the agent is free. What makes it worse is that there is no "althost" mechanism for the winbuild farm. A job in the queue is assigned randomly to any server on the farm. If it happens to be assigned to a busy agent, it will not proceed even if there are free agents available elsewhere.

6.7 Unix Packaging

XXX

• What is staging?

• Why there are several packaging steps - some done on cross compile hosts and others on native?

6.8 Windows Packaging

XXX

7. Miscellaneous Build System Changes

7.1 libproduct vendor library

Both NetApp and IBM builds happen as one. To achieve this we did the following:

• Moved libdfm/product.[ch] into a standalone dynamic library called libproduct

• Made product.h the sole interface for supplying vendor specific information at runtime.

• Removed all compiled-in vendor constants in the code and use libproduct to determine the vendor strings to use at runtime.

• Modified the packaging steps to install the correct library.

7.2 Client bundling

To include a particular version of the client software, modify the "include\_products" file at the top of the DFM source tree to include the right version. For example, this will bundle version 2.2 that is in /x/eng/rlse/sumo/2.2/:

sumo:2.2

The bundling only happens in the official nightly, special or promote builds. Private developer builds do not bundle any client software. The build system is designed to handle the absence of the client software gracefully.

7.3 No ctod

We removed the script 'ctod' from DFM. 'ctod' was a perl script that relieved us from the responsibility of writing forward function declarations. It slowed down the build, especially on Windows, probably because of high perl startup time. We added all forward declarations back into the source files and removed 'ctod'. We must write forward declarations ourselves from now on.

7.4 Relocation of 3rd party software

We relocated Apache, openssh and Putty to the champagne\_shared depot to avoid building these fairly sizeable pieces of software in every nightly build and every developer's workspace. These components are now brought over to the DFM workspace as prebuilt binaries at "src/champagne\_shared".

7.5 Optimization and debug symbols

We use compiler optimization to build the release binaries and include full symbols in them. On Windows the symbols are in the .pdb files. On Unix, the symbols are in the non-stripped binaries in the build/{platform}/bin directories. Before packaging the unix binaries, however, the symbols are stripped. Since optimization is turned on, it is a little more challenging to follow the code in a debugger. The availability of full symbols makes this manageable.

8. Build Time Code Generation

8.1 libsoapd - Web Service for ZAPIs

The build process for the libsoapd library includes elements of code generation based on the defined ZAPI interfaces for DFM. It includes an element of a custom generation program to generate WSDL from zapidoc, gSoap tools to generate .h and .cpp files from WSDL, and some perl scripting to adjust elements of the gSoap generated code to integrate the proper dispatching of the API calls internally to ZAPI handlers.

The zapidoc to WSDL conversion is accomplished with a generation program based on the Autozapi Java Binding function, the source code of which can be found in the /depot/prod/newportNews Perforce depot. For purposes of supporting the DFM build, a specific tested build of this generator iis delivered in the libsoapd tree.

The generator requires some common 3rd party utilities, which are sync shared in from the /depot/prod/netapp\_ivy\_repo Perforce depot. This is done in order to leverage the common Black Duck identification of this libraries done from this depot. This is all accomplished via the standard SharedComponents.cfg mechanism.

9. What's Next

There is still much to improve in the DFM build proces. These are some notable challenges:

9.1 Library re-organization

Our library organization is one horribly entangled mess. One telling sign is this:

$ ./make\_dfm --linux ping

chdir into /x/eng/buildw1/scratch/slam/p4/aw/main-03

lin: Fri Jun 6 12:06:17 2008: make\_dfm started

lin: chdir into /x/eng/buildw1/scratch/slam/p4/aw/main-03/build/linux

lin: Running /usr/software/rats/bin/rehost2.pl --sendenv \

--hostpat="LIN Platform3x" /usr/software/bin/gmake-3.81 -j ping

lin: > althosts: selected host "bldl24"

lin: [ 5%] Built target generated

lin: [ 5%] Built target libproduct

lin: [ 5%] Built target libnetapplm

lin: [ 5%] Built target libdfmauth

lin: [ 5%] Built target libsdapi

lin: [ 7%] Built target libndmp

lin: [ 11%] [ 11%] Built target zlib

lin: Built target libzapiinter

lin: [ 11%] Built target libcms

lin: [ 11%] Built target libsdclient

lin: [ 21%] Built target libldap

lin: [ 22%] Built target libadt

lin: [ 25%] Built target libxml

lin: [ 32%] [ 32%] Built target libdfmapi

lin: Built target libmonitor

lin: [ 37%] Built target libnetapp

lin: [ 44%] Built target libfilerapi

lin: [ 45%] Built target libadtpp

lin: [ 47%] Built target libnazip

lin: [ 50%] Built target libperf

lin: [ 52%] Built target libjob

lin: [ 55%] Built target libfaml

lin: [ 57%] Built target libconformance

lin: [ 58%] Built target libdatamgt

lin: [ 70%] Built target libdfmcore

lin: [ 78%] Built target libcmd

lin: [ 80%] Built target librbac

lin: [ 84%] Built target libmon

lin: [ 91%] Built target libevt

lin: [100%] Built target libdfm

lin: [100%] Built target ping

'ping' is the build target for dfmping. It is a simple binary that should only link in libdfm and libproduct. However, take a look at the log above and observe how many libraries were built. Almost 30. If the libraries were organized properly, the number should be closer to 5. In this particular case, there were several cyclic dependency paths that caused such a large number of libraries to be brought in. Some examples of the entangled dependency paths are:

• libdfm/burel.c includes libmon/mon.h, which includes libevt/event.h, which includes libcmd/fv.h, and libcmd depends on libdfm.

• librbac/rbac.h includes libcmd/obj.h, which include libdfm/rbac\_op.h, and libdfm depends on librbac.

• monitor uses libmon. libmon includes libevt. libevt has some cpp files in it which require libdfmcore. libdfmcore cyclically depends on libdatamgt, and both depends on libfilerapi and libdfmapi, which need libzapiinter.

We also have a long standing issue of using the non-thread-safe libcmd in all the daemons.

What should we do? Here are some possible answers:

• Resign to the fact that we cannot possibly keep track of all 50 libraries. Let's shrink the number to three - a thread-safe version for daemon, a non-thread-safe one for commands, and a common one used by both.

• Do nothing.

• Something in between.

9.2 Convert to dynamic libraries

We statically link almost all DFM binaries. We should benchmark the impact on build time and binary size if we were to switch to building dynamic libraries.

There are some challenges that currently keep us from trying:

• Windows DLLs by default export no symbols. We have to all dllspec to all functions we would like to export, or write a script to generate separate .def files that tell the linker which symbols to export.

• The MSVC linker cannot create libraries with cyclic dependencies. There is a way to work around it. See burt298907.

9.3 Remove unused #includes

We have been careless about what header files to include in the source files. Our favorite cut-and-paste reuse method leaves many unused #includes in our code. Coverity has an option to detect unused headers. We need to run that.

10. Useful References

• Champagne Depot Document Index

• DFM Development Guide for Windows

• DFM Development Guide for Solaris

• CMake Online Man Page

• GCC Command and Options

DFM Hitchhiker's Guide

NetApp Confidential

Contents

[hide]

• 1 Useful Links

• 2 Getting Started

o 2.1 1. Take Notes and provide feedback by editing this document.

o 2.2 2. Find out who your "buddy" (NetApp Mentor) is and make contact.

o 2.3 3. Follow the New Employee Guide

o 2.4 4. Mailing lists

o 2.5 5. Software Installation

o 2.6 6. Development Guides

• 3 Debuggers

o 3.1 UNIX

 3.1.1 ddd

 3.1.2 tui

 3.1.3 Using gdb on Red Hat Enterprise Linux ES release 3 (Taroon Update 3)

• 4 Coding Conventions

• 5 C and C++

• 6 Mapping project code names to products

o 6.1 Sumo

o 6.2 DFM releases

o 6.3 Acropolis

o 6.4 Data OnTap Releases

o 6.5 Spinnaker Releases

o 6.6 NetApp Filer Platforms

o 6.7 NetApp Host Agent

o 6.8 SAN host side software

 6.8.1 SnapDrive for Windows is Anvil

 6.8.2 SnapDrive for UNIX is Sawzall

 6.8.3 SnapManager for Exchange

 6.8.4 SnapManager for SQL Server

• 7 The DFM source tree

• 8 The DFM Build Daemon

o 8.1 The Build Daemon

o 8.2 The Tree Commands

 8.2.1 getstree

 8.2.2 dfm\_gettree

 8.2.3 deletetree

o 8.3 Sync'ing up your tree

o 8.4 Questions or problems with these commands

• 9 The DFM Build Environment

o 9.1 Parallel Make

o 9.2 Build Farm

o 9.3 UNIX build farm

o 9.4 Windows build farm

 9.4.1 Caveats

o 9.5 Treebuild

• 10 Testing Your Code in the RTP Lab

• 11 DFM Portion of the Design Depot

o 11.1 Project Web Page Directories

o 11.2 Roadmap Directories

• 12 Specifications

o 12.1 Adding a New Specification

o 12.2 Functional Specifications

o 12.3 Design Specifications

• 13 Mergeing Trees in US and India

• 14 The "r" command

• 15 Naming DL's

• 16 Updating 'man' Pages For the DFM CLI

• 17 Don't use 'p4 submit' on Windows

• 18 Running Pre-Checkin Smoke Tests Using DFMANT

• 19 Adding More Stuff To This Guide

Useful Links[edit]

NetApp HHG

New Employee Guide

Getting Started[edit]

1. Take Notes and provide feedback by editing this document.[edit]

2. Find out who your "buddy" (NetApp Mentor) is and make contact.[edit]

3. Follow the New Employee Guide[edit]

4. Mailing lists[edit]

Use majordomo to subscribe yourself to at least the following mailing lists (in addition to thosee recommended in the engineering hitchhiker's guide).

Your group's DL

Your product development group's DL (e.g. dl-protmgr-server-dev)

A child DL of dl-dfm-eng, such as dl-dfm-dev or dl-dfm-qa

Select the right DLs by looking at the list descriptions on the DFM Document Depot web page under "Contacting the DFM Team".

5. Software Installation[edit]

As of 5/29/2008, if you wish to build DFM on your desktop, you'll need a copy of MS Visual Studio .NET 2005 Professional Edition and CMake 2.6.0.

Hummingbird Exceed (X-Windows Server for your PC)

You can find Exceed and MS Visual Studio 6.0 on [file://eng.netapp.com/software \\eng.netapp.com\software]

For installation instructions related to VC 6.0 look at DFM Windows Development.

6. Development Guides[edit]

Try out the Windows Development Guide and the DFM Build Guide

Debuggers[edit]

UNIX[edit]

As of 5/29/2008 we are using gcc 4.2.3. This requires gdb version 6.0 which you should be able to invoke on any development machine by typing gdb.new.

If you like GUIs you have two options:

ddd[edit]

You can invoke ddd on both solaris and linux and it has most of the point and click functionality you could desire.

To invoke ddd use

ddd --debugger gdb.new

tui[edit]

On Linux you have one more option the Text User Interface of gdb. This interface uses curses rather than X or QT that ddd does.

To invoke the TUI of gdb use

gdb.new

and at the command prompt

(gdb) -

Using gdb on Red Hat Enterprise Linux ES release 3 (Taroon Update 3)[edit]

Use /usr/local/bin/gdb.es to debug on Red Hat Enterprise Linux ES release 3 (Taroon Update 3).

Coding Conventions[edit]

The DFM team is very strict about its coding conventions unlike most of the other groups within Data OnTap, and has managed to maintain a self-consistent style of code across all of the major sub-systems. Before you write code you should familiarize yourself with those conventions.

C and C++[edit]

The DFM team has chosen to write the new DFM apiserver in C++ (the process known as dfmserver). Our decision to use C++ rather than Java was based that we felt it was the least disruptive choice and got us the biggest bang for the buck. We also believed that the DFM code base would have to use components from the Filer and that those components would either be C (from Data OnTap) or C++ (from Spinnaker). This does imply that for a long time we expect C and C++ code to co-exist.

As a general rule of thumb, if the code is expected to live in the dfm cli then it probably should be written in C. If the code is expected to live in the API server then it should be written in C++.

Mapping project code names to products[edit]

Sumo[edit]

This is the internal project name for the DFM thick client. It is both the name of the framework and the name of the client.

DFM releases[edit]

These are all named after liquours.

Acropolis[edit]

This was the project name for the original infrastructure project that produced the first version of the dfmserver.

At some point we thought about having follow on releases named after pieces of infrastructure but have since decided that that makes little sense.

Data OnTap Releases[edit]

All Data OnTap releases are named after beer.

6.5 is Scrimshaw

7.0 is Anchorsteam

7.1 is TsingTao

Spinnaker Releases[edit]

Beepy named these after Tops in a game. Rumour has it that after the Convergence then we will go back to beers.

The releases are:

Tricky is the first Spinnaker with Data OnTap functionality

Shiner is the almost converged product between Data OnTap and Spinnaker

NetApp Filer Platforms[edit]

FAS2x0 are called shrunken heads. FAS250 is tsantsa FAS270 is jivaros

Exelesior is the next high end filer platform based on opteron hardware

Deux is the next mid-range filer platform.

NetApp Host Agent[edit]

The NetApp Host Agent goes by the name "Caffeine" or sometimes "SAN agent" or "SRM agent".

SAN host side software[edit]

The SAN team has a set of products that simplify LUN provisioning on the host side.

SnapDrive for Windows is Anvil

This software is used to provision LUNs on a windows box.

SnapDrive for UNIX is Sawzall

This software is used to provision LUNs on a UNIX box and is produced by the NANE organization.

Although Sawzall and Anvil nominally do the same thing, they have a different CLI, a different code base, and support different functionality.

SnapManager for Exchange

This software is used to provision Exchange using SnapDrive for Windows. This software knows about Exchange and how to setup the entire exchange database on NetApp gear. SnapManager \*uses\* Anvil.

SnapManager for SQL Server

This software is used to provision SQL server using Anvil. This software understands SQL server and understands how to setup the entire database on NetApp gear.

The DFM source tree

The DFM source tree contains all of the pieces necessary to build a release image for a DFM tree. However, the Champagne depot does not. The DFM source tree is constructed by copying from other depots source files and binaries that are required to construct DFM release.

We have a great FAQ that describes much of that information.

The DFM Build Daemon

The Build Daemon

DFM Build daemon is a daemon which runs periodically(every 10 minutes). It checks whether any new code has been checked in, and if so, it syncs a tree to the new code, and runs a treebuild(src/treebuild). which comprises linux, solaris and windows build.

If a build fails, it files a burt, and notifies everyone who has checked in code since the last successful build. If a build succeeds, it updates an "ok" link, indicating that this is the latest good build for that codeline and close the burt that was previous filed.

There is a separate build daemon for main and every active release codeline. For a list of available daemons for use with gettree, do 'dfm\_gettree -a'.

The Tree Commands

There are tree commands: getstree, dfm\_gettree and deletetree. They are designed to prevent you from ever having to run "p4 sync" and wait for the world to recompile.

getstree just checks out a copy of the source using a scratch directory in such a way as to allow you to use the Windows build farm. dfm\_gettree use the results of builds from build daemon to provide you with an up-to-date, precompiled copy of the source tree and its objects. deletetree is used to delete the getstree and dfm\_gettree clients.

The goal is to reduce the cycle time to build an initial client or bring an existing client up-to-date, and to isolate users from changes that don't affect them while giving them immediate access to changes that do.

getstree

getstree creates a directory on a shared scratch drive. This volume is set up so it can be mounted by the Windows Build Farm, which allows you to compile Windows code from a Unix system.

Usage:

$ cd ~/p4/dfm

$ getstree aw:champagne-kakia

09:08:48 ========================= getstree ==============================

09:08:48 > getstree aw:champagne-kakia

09:08:48 Gathering setup information...

09:08:48 User smoot

09:08:48 Site svl

09:08:48 NTBF WinFarm /u/build

09:08:48 VolMap /u/build/NTBF/ntapSVLscratchVols.map

09:08:48 Log File /u/smoot/getstree.log

09:08:48 Product aw

09:08:48 Codeline champagne-kakia

09:08:48 WinServer CfgEE

09:08:48 Codeline Builds Mixed

09:08:48 P4 Client smoot:wf:champagne-kakia

09:08:48 P4 Root /x/eng/buildw1/scratch/smoot/p4/aw/champagne-kakia

09:08:48 P4 View //depot/prod/champagne/champagne-kakia/...

09:08:48 P4 Opts noallwrite noclobber nocompress crlf unlocked nomodtime normdir

09:08:48 P4 Change Level Head

09:08:48 Creating P4 Client: smoot:wf:champagne-kakia

09:08:49 Sync'ing P4 Client: smoot:wf:champagne-kakia

09:08:50 Scratch Tree Creation Successful in /x/eng/buildw1/scratch/smoot/p4/aw/champagne-kakia

The line about P4 Root is important. I create symlinks from somewhere in my home directory to that tree:

ln -s /x/eng/buildw1/scratch/smoot/p4/aw/champagne-kakia ~/p4/kakia-gt

Now you can cd into that tree, run sync\_shared, make\_dfm, Perforce commands and winbuild.

Note: In the spring of 2010, the codeline name conventions changed. You used to just be able to run getstree champagne-freyareloaded. Now you need to prefix the codeline name with a product name. For us, that means you put "aw:" at the front (e.g. getstree aw:champagne-kakia). You won't need that for older codelines (up to champagne-juno and releases up to 4.0).

dfm\_gettree

dfm\_gettree is a program which creates a build tree in which all the object files are hard-linked to the latest successful build daemon tree. Clients created by gettree contain hard links to the build daemon trees which are shared among users. For this reason, it is very important not to change permissions or modify files directly. To work on a file, you must always use “p4 edit” to modify it. Otherwise, you can damage the build daemon and clients belonging to other engineers.

Usage:

dfm\_gettree //<codeline> <target-dir> <p4-client-name-to-use>

Example:

dfm\_gettree //Rchampagne b999999 ranade:BTC:b999999

deletetree

To free up space for ongoing development, please remember to remove your tree with this command once you've finished. If you have generated built files or binaries that own by root, use 'sudo deletetree'. Sudo use may not be neccessary especially in recent codelines as the need for root has been eliminated.

Usage: deletetree [-f] <my\_gettree\_dir>

Example: deletetree -f b999999

Sync'ing up your tree

If you have worked in the same tree for a long time, so you probably have missed others check-ins and your tree is probably out of sync. In that case don't use "p4 sync" command because it has no knowledge how to handle shared components.

Instead, go to the top directory of your tree, and use the SharedComponents\_sync command to bring your tree up-to-date.

Questions or problems with these commands

If you have any requests, problems or questions related to the build daemon, please contact [1].

The DFM Build Environment

DFM unlike Data OnTap consists of a set of distinct libraries that are linked together. As a result building dfm (other than the initial) build is rather quick since you typically only have to re-build the library you are working on and do a re-link.

There are plenty of gotchas, however, related to the nature of the cross-platform challenge of building for two different OS's UNIX and Windows.

We try here to enumerate some of the ways to minimize the pain associated to a build.

Parallel Make

For both Caffeine, and DFM UNIX builds it is possibel to use the gmake facility to start concurrent builds using the -j flag.

make -j

will start as many parallel builds as the machine can handle.

make -j N

will start N parallel builds on the machine.

Build Farm

NetApp has two distinct build farms to off-load builds from your desktop a UNIX build farm and a windows build farm.

UNIX build farm

The UNIX build farm can be used by typing the following at the UNIX command line

make solaris-all

or

make linux4-all

The make script will poke into a set of build configuration tools to find a machine that matches the right configuration (has the right set of tools/libraries installed) and do a build by going to that machine.

This build environment can be used to do complete builds of dfm much faster than trying to log onto a compute server and firing off a build there.

Windows build farm

The windows build environment requires a different scheme and has the disadvantage that it does not provide immediate feedback, sometimes hangs, but is the only way to test your code on the same kind of environment that the nightly build is created.

To use the build farm you must first have checked your tree out using getstree, and in the root directory of the dfm tree type:

winbuild

The script uses a configuration file called winbuild.cfg. One of the options that you can put in that configuration file is the "noclean" which reduces the build time by compiling only the changed files.

The configuration file has three lines

Codeline champagne-gayatri

WinCfg CfgD

Run Makeaw.pl

The "noclean" flag should be added to the Run.

The WinCfg line should say CfgD if you are building hennessy (DFM 3.1) CfgB if you building anything earlier than hennessy.

There are other winbuild options:

noclean

Do not clean the build tree first

nodebug

Do not build Debug objects and executables

norelease

Do not build Release objects and executables

nofixrc

Don't fix RC files. I honestly don't know what this does.

nopackage

Don't build installers. Building the installers is quite time consuming, so if you don't plan on using them, this is a big time saver.

netapponly

Don't build IBM binaries.

useincredibuild

Build using the IncrediBuild parallel build system. This typically doesn't work because of build dependency issues.

Any of these options can be put in the winbuild.cfg file, or specified on the command line:

winbuild -run Makeaw.pl noclean norelease nopackage nofixrc

Caveats

One thing to be aware of is that windows and UNIX permissions do not mix well, and the build farm runs tool runs as user "buildfarm" that is different from you. As a result you can not touch your source tree while the build farm is running. Furthermore, if you start a UNIX build in the same tree the build farm can get very, very, cranky as generated files (.d files in particular) start being clobber by the UNIX build.

Treebuild

The dfm source tree has a script called treebuild that will generate a complete build of all of your files on all platforms. The treebuild script runs the various builds in serial (solaris/linux/windows).

Testing Your Code in the RTP Lab

See this guide.

DFM Portion of the Design Depot

The DFM portion of the manageability website is maintained in perforce, as part of the engineering design depot. To edit files there, you'll need to create a perforce client that pulls in files from:

//depot/web/engineering/design-depot/appliance-mgmt/...

This means, the view for your perforce client might be similar to: View:

//depot/web/engineering/design-depot/appliance-mgmt/... //perryj:dfm-design-depot/...

Project Web Page Directories

The DFM project web pages can be found in the champagne subdirectory. Each release is in a directory with a number. For example, the grappa release (DFM 2.3) web pages are in the 7 subdirectory of the champagne champagne, and the hennessy release web pages are in the 8 subdirectory. The numbers increase sequentially from the first DFM release. The exception to this is the initial DFM release, which lives in the champagne subdirectory.

Roadmap Directories

The roadmaps are kept in the roadmaps subdirectory, except for the Sumo roadmap which currently lives in the sumo subdirectory.

Specifications

The specifications are written using MML (Mini Markup Language). The MML source is converted to HTML using /usr/local/bin/mml.

Adding a New Specification

To add a new specification for a given release, follow these steps:

1. If you don't already have one, create a DFM design depot client, sync up, and then switch to the appropriate subdirectory for the release (e.g. champagne/8 for the hennessy release).

2. Using your favorite editor, generate the initial draft of your spec. Remember to use MML.

3. Add the file to perforce using p4 add: p4 add -t ktext your-spec.mml (remember, this doesn't actually get added until you do a p4 submit). For design specs, we typically use a name like feature-ds.mml and for functional specs, we typically use a name like feature-fs.mml.

4. Edit the index.mml in the same release directory that contains your spec so that it can link to your spec. This typically means adding an MML doc macro to the file.

5. After you submit your perforce changes, a cron job executes on the web server that hosts the design depot, and converts the MML page to an HTML page. In order for this to happen though, you need to modify champagne/Makefile to generate the HTML file. To do this, find the Makefile macro that contains the file list corresponding to the release your spec is for. For example, the list of files in Hennessy can be found in the HENNESSY macro. Add the HTML file name for your spec (not the MML name). For example, for the audit logging specification I added audit-logging-fs.html

6. Verify that your changes work, and that your MML file is correct by generating the web pages for your release. This is done using champagne/Makefile. For example, while in the champagne subdirectory, you can use make hennessy to generate all of the web pages for the Hennessy release. Alternatively, you could generate all of the web pages for the DFM web site using make all. That of course takes more time. Or, if you want to save even more time, at the expense of a little more typing, you can use make <release-dir>/<your-spec.html>. In the case of the audit logging functional spec, that would be make 8/audit-logging-fs.html.

7. This step is optional, but can be very useful for previewing your changes. You might want to create a web subdirectory in your Unix home directory, and then create a symbolic link to your dfm design depot area so that you you can look at your changes prior to submission.

8. Once you are satisfied with your changes, do your p4 submit. In 5-10 minutes, the web site will be updated (the cron job executes every 5 minutes).

Functional Specifications

Functional specs are meant to completely specify the interfaces and functionality of a feature. The intended audience for the spec is QA, technical publications, and marketing. They need to be able to use this document as a reference to determine what tests to write, document the feature for customers, and verify that the feature will provide the functional required in the market.

The storage management team as of Janneau (DFM 3.3) uses a functional specification template

Design Specifications

Design specs are used to document what is needed to develop a feature in more detail. In DFM, they typically are written only for the more complicated features. In general, they will document complicated algorithms being proposed, database changes needed, or data structures that will be used.

Mergeing Trees in US and India

Suppose you have a tree in the US and you are now in India and need to continue making changes to the tree in the India, but want to not to do those changes over the wan.

The solution is not obvious, but here it is:

$ cd top-of-client-in-the-US

$ p4 cli\_snap > /tmp/SnapFile

$ cd top-of-client-in-India

$ p4 cli\_restore -n < /tmp/SnapFile

Please note, if you do the p4 cli\_snap on a solaris box, you need to do the p4 cli\_restore on a solaris box.

The "r" command

At NetApp because there a lot of time when you need to do something as root, we have a special command called "r" that does a sudo to root. The command is logged (so don't think you can do anything you want with no reprecussions) and is not enabled on all machines. Furthermore, some volumes have permission set so that even root does not work.

An example of it's usage:

$ r ls

Naming DL's

The DFM team has the convention that all dfm dl's should be named

dl-dfm-projectname

When creating dl's be aware of that...

Updating 'man' Pages For the DFM CLI

The Windows build doesn't currently run the perldoc conversion that converts the \*.pod files we edit in our source directories to html man pages. So after making your change to the \*.pod file, building and checking your changes, you should also copy the .html version of the man page into the src/web/man directory and check in that .html.

As an example, if you add new global options to the dfm command, you would then do the following on a linux build system:

$ cd src/cmd

$ p4 edit dfm.pod

$ vi dfm.pod

$ make man

$ [ view the ../../release/linux/man/html1/dfm.1.html file to make sure it's formatted OK ]

$ p4 edit ../web/man/dfm.1.html

$ cp ../../release/linux/man/html1/dfm.1.html ../web/man

$ [to make ibm man pages]

$ cd ..

$ make man VENDOR=ibm

$ [ view the ../release\_ibm/linux/man/html1/dfm.1.html file to make sure it's formatted OK ]

$ p4 edit web/man-ibm/dfm.1.html

$ cp ../release\_ibm/linux/man/html1/dfm.1.html web/man-ibm/

$ p4 submit