Distribution and Configuration System

Release 2.0

John P. Rouillard

January 2009

Contents

List of Tables

List of Figures

Chapter 1

DACS Introduction

What is DACS? DACS, the Distribution and Configuration System, is a tool for maintaining system configuration on multiple computers. It is similar to other CCM (computer configuration management) tools such as bcfg2, lcfg, puppet and the well known cfengine. However, it has some unique features that makes it more than just a program which pushes files to other hosts. It integrates:

- a host database
- a version control system
- an optional file generation system
- a file distribution and remote command execution mechanism

DACS started life as the config system (`http://www.usenix.org/publications/library/proceedings/lisa94/rouillard.html`) back in 1994 when it was presented as a paper at LISA. It's current home page is: `http://www.cs.umb.edu/~rouilj/DACS`. It was intended as an experimental platform to provide what I considered crucial components of a CCM system.
Despite being out of system administration for 5 years, I still got occasional reports of companies using Config. I was amazed by some of the (crazy) things people were doing with Config. This included managing VMS systems and a 5000 node network.
When I was rolling out a CCM system at my new job in 2005, I looked at the existing systems. They had minimal support for some of the key things I wanted in a configuration system:

- a database of information about systems that can be queried
  - to generate configuration files. E.G. get the addresses of the local NTP servers from the database and automatically insert them in generated ntp.conf files to eliminate the manual maintenance of those files when the servers change.
  - to generate lists of computer wiring
  - to generate lists of assets
  - to select hosts running a particular OS version

- ability to track changes to the configuration system
  - Who did what change
  - When was the change made
  - What files were changed and how
  - Why was the change made
- the ability to generate files in response to configuration changes to keep them in sync with changes to the network
- the ability to quickly determine what services were lost if a host failed

As I write this in December 2008, some of these items are present in most of the CCM systems (e.g. hosts are allowed to have additional information associated with them), but not all of the functions are available. For example, version control still has an add-on feel in most of the other systems rather than being a core part of the functionality. Also, performing database queries from outside the tool is still somewhat clumsy.
The key element in DACS is the use of the database as the repository of configuration knowledge. Once that is present it makes a lot of other tasks easier and possible.

1.1 Should you deploy DACS?

DACS is one of many CCM systems. It was not meant as the end-all and be-all of CCM systems. It was designed to make managing 100 systems by one person tractable.
I have successfully used it for 1-200 hosts including a mix of Cisco and Unix/Linux based systems. DACS should work well for 300 machines per installation. I have a report (from 10 years ago) that it has been pushed to 5000 systems; the techniques used were inventive and it worked for them allowing some really great functionality (see Renumbering a Network and a Caution About Automation section 7.12).
With more than 300 hosts the reports get difficult to read and it is difficult to obtain a clear view of the system configurations. This is especially difficult when there are a lot of anomalies between the managed systems and the CCM system. This problem is not insurmountable, but is an issue.
I have also used it in environments with 10 different operating systems and I expect it would work for many more.
I have used DACS/Config with 1 to 10 other administrators. DACS locks the configuration tree while an update is being pushed to maintain consistency. Because of this, contention for the shared tree can cause frustration as some admins are locked out. However, DACS does support separate configuration master trees section 8.1.4. You can assign one tree to each administrator to allow parallel testing and deployment. Using multiple trees does cause some temporary increase in variation across your managed systems.
While I and a few others have used it successfully, it is not as polished as other tools. The documentation is a bit spotty in places. Also DACS is not a monolithic tool. It incorporates multiple tools which implement the various components. These components are integrated by a wrapper. Because of this, the implementers (although not all the users) must be familiar with or be willing to learn about the various tools:

- Perl
- Bourne shell
- text file database tools
- svn or cvs
- gnu make
- filepp
- rdist

The advantage to this is that DACS is flexible enough (if you know Perl) to replace "gnu make" with your favorite build tool (say ant or cons). If git is the preferred version control system it can replace svn/cvs. This can be done with some Perl script editing. Filepp is used for macro expansion and file generation from templates, but it can be replaced with your preferred tool. For example, in Config templating was done with sed scripts. The native 'database' can be replaced with something SQL based section 3.2 if you desire. The Perl script `Rdist` is used to integrate all these parts. It can be modified to suit the way you want to work.
There is a saying that a single admin can destroy a machine, but to destroy a site requires a single admin multiplied by a CCM system. In this respect DACS is no different from any other system. However DACS provides some safeguards if you wish to implement them:

- the version control system can implement safety checks and refuse to allow the check-in of bad configurations or syntactically incorrect files or scripts.
- the build system can run verification checks as part of the distribution and will not push any changes if the verification fails.

Configuring a host should be done by modifying it's entry in the database. This way the database becomes the ultimate repository of all knowledge about that host. You don't usually configure DACS to push a file to a specific host. Instead DACS provides classes (of hosts) which are used to define the target machines for a file. The classes are automatically defined from information contained in the DACS database. This abstraction makes it possible to determine the configuration of a host by consulting the database.
You can choose to use a direct file to host mapping if you wish, but it makes gathering information more difficult as your database is no longer the ultimate repository of knowledge. The knowledge about host configuration is not in the database, but in the file that has the file -> host mappings.
You should not use DACS (or any CCM system) if you are just barely able to edit a hosts file successfully the first time. However DACS is the right fit if you have mastered running a single system and can plan out the steps needed to expand that to multiple systems.
For example if you understand the following steps for deploying the network time protocol at your company:

- set up the top level NTP servers that all the rest will receive their time from
- generate the ntp.conf files for the top level servers

- generate the ntp.conf files for the hosts that will use the top level servers
- open firewall ports on the servers to allow NTP to pass

you will probably be able to successfully use DACS.
Because of the database, the feedback I have received is that you really have to understand what you are trying to implement and lay it out before you start putting it into DACS. This level of planning before implementation is not enforced by DACS, but you will waste a lot of time if you don't plan out what you are doing. Lack of planning will force you to keep revising the setup or worse, start embedding configuration information in places other then the database. When you change the configuration (and it will change) you need to search all the files in the CCM system to find the embedded configuration information.
The effort of implementing and working within any CCM system is very visible, while the savings are hidden. The effort includes:

- the learning curve to use the system
- in CCM systems changes are generalized to allow them to apply to multiple systems rather than just one system. This makes deploying making large scale changes so much easier than logging into 200 individual systems and editing the same (well hopefully it's the same) config file on each system.
- wasted time while files are generated and pushed. While this may appear to be unproductive time, the time can be used to document the changes in the ticket or do something else while waiting. Compare this to editing 200 files which looks like work, but is really just busy work.

while the savings include:

- fixes are implemented once and applied to a class of machines. Hence you aren't continually fixing the same problem wasting your time and the time of the people who need the fix to do their work. The effort is expended once per DACS installation and not once per machine.
- reduced impact due to changes. Standard configurations allow changes to be tested more thoroughly and the DACS database can be used to look for host configurations which may cause a problem allowing test prior to deployment on the edge cases.
- faster deployment of new/replacement systems. Because all the unique host info is in DACS, it can be pushed out to a new system to replace a broken one.
- reduced archeology - when making a change on manually managed systems, you often have to do a fair amount of digging and interpretation to understand how the system is set up and how it is supposed to operate before you make a change. If you have 100 systems it is possible you may have to do this 100 times prior to making a change. CCM systems by their nature have a tendency to reduce this variability (by using one file for multiple hosts). Also by gathering all the files together it is easier to compare them when there are multiple files. Also with a VCS system, the log messages supplied when the changes were done provides the context needed to understand the changes.

Many of these wastes of time (archeology, breaking systems with changes, fixing things over and over again) are considered "business as usual" at many locations and the cost due to this is never identified. However the time spent properly implementing a CCM system which reduces these (and other) causes of waste is very visible. I would love to have solid statistical evidence and case studies proving CCM systems result in performance improvements, but sadly I do not. I think the biggest issue is that creating these studies requires time, planning and commitment which are in short supply when you're running in firefighting mode most of the time.

1.2 Why you should deploy DACS

Now that we have discussed some of the reasons why you may not be able to deploy DACS, let's discuss some of the features of DACS when properly deployed:

- Accountability - the person responsible for each change to the system is recorded by the version control system.
- Automatic documentation - documentation in the form of a check-in log message is requested at the time the files are checked into the version control system. The version control system can require a trouble ticket identifier linking this check-in to the requirement.

  As a second form of documentation, using DACS as the ultimate authority of what should be running where, the documentation (in the form of database settings, files in the CCM system) and the systems are always in sync. DACS will warn you when the docs don't match reality.
- Quick repair - if a system is misbehaving, DACS can be used to verify that the system is running the proper configuration. If it is not DACS can be used to impose the proper configuration. This cuts down on the time to repair by fixing the problem, or eliminating possible causes.

  In addition the VCS underlying DACS can be used to search for operation impacting changes. So rather than looking through dozens of files on the misbehaving host, DACS can quickly narrow down the search to files that have changed recently. Even better, it can narrow down the field to specific changes within those files that caused the problem.
- Reduce complexity/variation - you can get reports on the number of files pushed to a machine and the number of tokens that defines configuration settings is directly proportional to the complexity of your environment. In the database you can see how complex your configurations are getting and how many variations of configurations you have. You can use these as metrics to try to reduce the amount of variation. E.G. you have six different copies of /etc/httpd/conf/httpd.conf being pushed. Why not see what the differences are and reduce it to three or even to one copy of the file. The goal is not to run 100 systems but to run 1 system 100 times as this leads to less documentation, training and confusion.
- Faster modifications - because all the files are identical or similar across all machines, it helps re-enforce best practices for setting up software. As a result there is:
  - one instruction manual/training set on how to do it

  - less time spent trying to understand some novel configuration before it is modified. The configuration is standard and hopefully documented, so changes can be accomplished quickly.
  - better testing as the database can identify different host configurations which should be used for testing. This should reduce deployment problems caused by the change allowing them to be discovered during testing.
  - less manual work by using templates and generating files rather than editing 200 files. Instead edit one template file and have the system build the 200 files. This frees you to have coffee, answer email or write documentation while the system is building the files. Plus it reduces variability as the files all follow the same template.

- Increase predictability - rolling out changes shouldn't result in surprises. By having a standard platform, unintended interactions are minimized and can be found during testing rather than after deployment.

- Oops recovery - since all changes to the configuration tree go through the version control system, you can always roll back a bad change and push the older (i.e. working) configuration. Then you can use the difference tool in the version control system to see what you changed and understand what actually went wrong.

- Why is it configured that way - have you ever looked at a configuration file and thought, "well that's wrong" and "fixed" it? Then three months later discovered it was right and you now have a week of repair/recovery. With a version control system enforcing the requirement to have ticket information in the check-in message, you can relate changes in the configuration to tickets and discover the cause for the change before you undo it.

- Delegation - finding a system administrator for every configuration change isn't always the way to go. Wouldn't it be nice to be able to allow a user to make changes in a controlled way? Using the version control and build systems you can delegate changes to others. If things break, you can see what changes were made. No more cries of "but I didn't change anything".
  Using the version control system you can delegate an entire file to one or more users to change, or you can have them edit a control file whose changes are validated and merged into other configuration files by the build system. Further details can be found in Delegation of access to a non-admin user section 4.4.4.

None of these features comes without a cost, however, I believe the features outweigh the cost. The biggest difference between DACS and other system is the explicit support for a "database". The database can track equipment even if it is not managed via DACS. The database can be mined to extract a lot of information which would otherwise be scattered throughout files in the DACS tree. For example:

- what is the IP address for a host?

- what services does a host run? (Used to determine what files should be pushed, what firewall configuration settings should be enabled, what services should run on boot etc).

- what are the attributes of a host? (What patches does it receive, does it have a 3ware raid card so the control program for it should be loaded on the system etc.)

plus the database can provide answers to the questions:

- how many unique configurations of systems do I have
- how many different configurations of this service do I have deployed
- how many hosts are running a particular OS
- what hosts are running services exposed to the Internet

The database is meant to be the ultimate repository about information for each system, from Ethernet address to whether it allows ssh access from the Internet.

1.3 The Components

DACS consists of 4 basic components:

- Database - defines properties and capabilities of the hosts. It is sort of a prototype configuration management database (CMDB) from ITIL.
- Source code control system (currently subversion with some support for the older CVS) tracks all changes to the database, build system and distributed files.
- File build mechanism based on gnu make(1). It can use database info via the `dbreport` command to extract IP addresses, host names and other host attributes to build custom files (perhaps using filepp(1)) for hosts. Using this is optional but recommended, as it reduces variation between files and allows for reuse of configurations between machines.
- File distribution/remote command execution mechanism based on rdist(8). A control file is provided to define labels: a set of files and remote commands to run as a unit.

and includes the commands:

- Rdist - main Perl script wrapper
  - updates, generates and pushes files to the managed systems
  - supports host verification before files are distributed. This makes sure the machine matches the database values (this is done by the HOSTVERIFY script). If the verification fails, the host *not* updated.
  - determines the default distribution targets by searching for labels in it's control file (Distfile) rather than from the list of top level directories (as in Config).
  - supports verification which doesn't change any operational files and is used for generating a report on what would be pushed if a normal distribution run was done.
  - allows specialized -verify targets to run which verify configurations that are not file based or can be dynamically reconfigured. For example firewalls or a windows registry.

  - optionally enforces distribution to a limited set of hosts (i.e. a test network) to prevent accidental installs of files on production hosts.
  - provides a mechanism for determining and distributing the Rdist command line to other systems. This is used to set up a hierarchy of Rdist servers to provide load balancing, redundancy etc.
  - permits supplemental targets to be defined via .targets files (target override).

- dbreport - a Perl script for querying and validating the database. (For those familiar with the Config system, this replaces class_gen.) It:
  - provides regular expression based host selection
  - allows host selection using any host property in the database
  - provides simple query/report formatting capabilities
  - is used to generate reports such as asset lists and wiring tables
  - provides data validation via regular expressions to detect typos in any field.

- cleangrep, cleanfind - allows searching of files under subversion control to identify files affected by config changes. They ignore files in the working tree not checked into subversion, including generated files.

and uses the following external tools:

- subversion or CVS - version control system
  - provides accountability for changes
  - provides authorization for users to change files. This allows the right to change a file to be delegated to particular users.
  - allows rollback of changes to a known good state.
  - implements safety checks on files to prevent broken files from being checked in.
  - records documentation on each change entered
  - optionally verifies each change message to make sure required elements are present (e.g. ticket numbers)
  - optionally distributes change messages via email or other mechanism to notify other and implement change review process.

- gnu make - acts as the build system for DACS
  - provides the ability to generate files from templates or other mechanism
  - sets ownership/permissions/group for files which need to be different from the default settings.
  - implements validation and verification checks which halt distribution if they fail.

- filepp - a macro pre-processor written in Perl. Similar to cpp but better and simpler to use than m4.

  - used with an extensive set of macros for rdist to make specifying common operations easier
    * saving of N copies of a file on the remote host
    * force execution of commands on remote hosts
    * rule to report the target (remote) file name
    * diff a remote file with the config copy
    * others (see the automatically generated macro documentation)
  - Can be obtained from `http://www.cabaret.demon.co.uk/filepp`.

- rdist - distribution program which is supplied by most Unix vendors. Version 6 of rdist (which is the current release at the time this was written) is preferred.
  - provides file distribution and remote command execution functionality
  - an intro to rdist and its syntax can be found at `http://www.benedikt-stockebrand.de/rdist-intro_e.html`

- sed - a program used for file processing. The one installed on your operating system should work fine.

- sudo - used to provide elevated privileges while being able to track the actual user performing the execution via the SUDO_USER variable. This is technically optional and can be bypassed by using `su` and setting the SUDO_USER variable manually, but the use of sudo is strongly encouraged.

- robodoc (optional) - extracts specially formatted comments from source files into mtml, text and other forms of output. It is used to extract the documentation of filepp macros, makefiles, firewall rules etc.

1.4 DACS Documentation

The DACS documentation is split into multiple files. The files are of three types:

- Basic introduction and concepts
- Training Docs
- Reference

I suggest reading the basic introduction files first then the reference documents in the order given below.

1.4.1 Basic introduction and concepts

Are covered by this document/chapter (section 1) which introduces the basics of DACS and the tools you need to implement it.
To see DACS in action as it were you want to read the DACS usage section 2 document/chapter. It discusses using DACS and hopefully shows how the concepts described in section 1 fit together and are used on a day to day basis.

1.4.2 Training docs

The training docs are some computer based training lessons I developed.
The CbtDacsDbreport.txt file provides a tutorial on using dbreport for generating database reports and getting information from the database.
The CbtDacsFirewall.txt is a tutorial on the iptables firewall generating system implemented in DACS and included in the distribution. They are provided as seperate files in the DACS distribution and are written in TWiki markup language.

1.4.3 Reference section

The reference section is composed of the majority of the documentation and is composed of one section for each of the 4 basic components:

- DACS database section 3 - all you want to know about the database and generating reports from it.
- DACS version control section 4 - using a version control systems (primarily subversion) with DACS.
- DACS build section 5 - using the build system with DACS
- DACS distribution section 6 - distribution of files and commands with DACS

The DACS examples section 7 chanpter/document provides a set of examples ranging from simple to complex. The examples provide more detail on the information presented in DACS usage section 2 as well as provide examples for new tasks.
There are advanced topics such as using multiple DACS master for redundancy and load balancing.

- Dacs Advanced section 8 - advanced topics

In the appendix is information about using the DACS distribution to create your own DACS installation.

- DACS appendix section 9 - appendix

Lastly there is the glossary of terms:

- DACS Glossary section 10 - glossary of terms

1.5 Getting Started

To get started with DACS:

- Read this chapter
- Read Using Dacs section 2 to get a feel for how things fit together and the types of operations you will be doing.

- Install the required tools on your system (if you run Linux most of these will be prepackaged) listed in the third list of section 1.3.
- Then visit the Appendix section 9 for instructions on importing the repository for testing and creating a working copy.
- Add some hosts to Config/database/db as you read the "database" system section 3.
- Create a Config/distfile/distfile.base while reviewing Dacs distribution section 6 and use `make files-report` to see what files would be installed. Maybe even set it up to install a file in /tmp so you don't make any operational changes.
- Check in your changes and create a master repository at /config following the directions in Creating DACS CCM master trees section 9.4 (Use the file:/// access method is you haven't set up an ssh based repository.)
- Set up your client host(s) following one of the methods in Setting up ssh access from the master to clients section 9.2.
- Try distributing to a client.

The homepage for DACS (`http://www.cs.umb.edu/~rouilj/DACS`) will provide other resources to help you get started.

1.6 Acknowledgments

I would be remiss if I didn't thank the people who contributed to DACS over the past 15-20 years.

- Rick Martin - Sysadmin Computer Science department at the University of Massachusetts who provided the basic idea and my editor for the original Config paper.
- Tom Bechard - System administrator where most of the initial development for Config occurred.
- Nerayan Desai - for some in-depth discussions of CM and how to improve performance.

Anything that makes sense in this documentation can be blamed on my reviewers:

- Darlene Choontanom
- Devdas Bhagat <devdas at dvb.homelinux.org>
- Nate McKervey

while all the errors are my responsibility.

Chapter 2

Using DACS

This section describes at a high level working with DACS on a daily basis. It glosses over the details that are covered is subsequent sections and provides a framework for how you interact with DACS.
You should read "The Components" section of DacsIntro to be familiar with the names and roles of the tools that are used in the following sections.

2.1 Use case troubleshooting a problem

You receive a report that Jane can't log into the web server. What do you do? If you are using DACS, you have a few things you can do before you log into the web server and start searching. First run `/config/Rdist -v -m webserver.example.com` to see if somebody made an unauthorized change that broke the service. Simply reverting unauthorized changes fixes many issues quickly with almost no effort. (Well ok, you have to expend effort to find out who changed things, but at least the users aren't impacted.) Let this run and move to the next troubleshooting step.
Since this problem deals with the web server, it would be nice to know if anybody changed the configuration on the server. If the server config is managed via DACS, using:

```
cd DACS/httpd
svn up
svn log -v |less
```

will give you a report of all the changed files and the log messages associated with the changes. You notice:

```
------------------------------------------------------------------------
r1034 | rouilj | 2008-12-13 18:54:31 -0500 (Sat, 13 Dec 2008) | 22 lines
Changed paths:
   M /repository/Config/work/httpd/conf.d/authenticate.conf

ticket:10233 Modifying to authenticate against the L3DAR ldap server
used for external authentication rather then the L1DFS ldap server.
```

Hmm, the change occurred three days ago and Jane said she could log in last Thursday. Well now we have something to investigate since the `/config/Rdist` run reported that everything was as specified in DACS.
Go to rouilj's office and tell him about the problem. He looks at the L3DAR server and realizes that Jane's info hadn't been copied from the L1DFS server because she wasn't listed as a user. He completes the operation to authenticate Jane and she is now happy.
Now let's run through a scenario where DACS isn't in place.
First knowing that the authentication comes from LDAP, you log in as yourself and verify that the authentication is working, for you at least. Since Jane sent email, you know she can log in, so that seems to rule out an LDAP issue.
You now log into the web server and look for recently changed files. You discover that /etc/httpd/conf.d/authenticate.conf has changed recently. So you view the file and start going through all the various authentication rules. Then you realize you don't know what area of the web site Jane was trying to log into. So you call her and find out. Then you start going through the authenticate.conf file looking for the rule that controls the area Jane was logging into.
Ok, you find it, but it doesn't look right. So what do you do?

1. change it back to what you think it should be (possibly breaking things in a different way).
2. start asking around to try to find out why authenticate.conf changed.
3. look at a backup of the file from last Thursday (the last day it was known to work). Maybe the change in the authenticate.conf file is correct and occurred before last Thursday.
4. look at the sudoers and su logs to see who may have changed the file between last Thursday and today.
5. shuffle the ticket off to somebody else.

While option 5 is looking very tempting, you decide to send an email to all of the admins. After 20 minutes you get back an email that indicated that the rouilj guy may have made a change. So you go to his office and sure enough he made a change. But he is old and getting senile and doesn't remember what the setting was before he changed it. So off to the backups to figure out what was working before. Now with the info in hand it is an easy matter to make the changes on the L3DAR to authenticate Jane.
But look at all the extra work and time spent solving this.
Now some people may say:

> yeah of course that was easy he documented in the log message all the info he needed. Most people never do that.

Well that's ok. Running `svn diff -r 1033:1034` will generate output that shows you the before (working) and after (not working) configurations in unidiff format. So even though the log message may have been only:

```
ticket:10233
```

You still have the information needed to fix the problem.
(In this scenario, the entry of a ticket number is enforced by the VCS, so every change has to have at least that bit of info.)

2.2 Use case recovering from a dead host

Your monitoring system reports that the host box01 is dead. You go over and power cycle it and nothing happens. No POST, nothing indicating any sign of life.
Now what do you do? Well you call hardware support, but you still need to restore any services that were running on the box so that others aren't impacted by the death of box01.
Turning to the DACS database the entry for box01.example.com:

```
machine = box01.example.com
cluster = site_mia1 operations production
disk = WD5000YS(500B) WD5000YS(500GB) WD5000YS(500B) WD5000YS(500GB)
os = Centos 5.1
services = NTP1
services = APACHEINT SUGARCRM TWIKI_DEVELOPMENT
uses = RAID3WARE_8006
```

So we know that box01 was:

- Running one of the top level NTP servers (the name (NTP1) indicates that it has redundant backups at NTP2, NTP3. We can verify this by searching the database.)
- Run apache whch supplies
  - the Sugar CRM application used by sales
  - the TWiki instance used by development

How do we know this information is up to date? Well this information controls the deployment and verification of the configuration.
DACS closes the loop between documentation and continual validation of that documentation. If it is out of date you have a report that is it out of date and can bring the installation in line with the documentation (or vice versa). Box01.example.com didn't have any anomalies in the overnight report, so the data is correct.
Also we know that to rebuild this we want to use the CentOS 5.1 operating system. We also know that the disks on the system are controlled/formatted using a 3ware raid controller model 8006.
Now notify the help desk that the server is down and the Sugar and Twiki instances are impacted. Tell them we are working to restore functionality. This way they aren't surprised and without information when the calls come in. Fortunately nobody has noticed yet, but it's only been 5 minutes.
So we know that we have two critical applications: Sugar CRM and TWiki that need to be restored, and one (NTP1) that can wait a bit. First thing is do we have another machine that can be used to do a disk swap? I need a 4 disk system with an 8006 (or at least an 8000 series controller). So I search the database using 'dbreport' for a development system that uses RAID3WARE_8006. No luck. We do have some other systems with the same card but they are production and they are running equally critical services.
As we look through the DACS tree we realize that Sugar CRM's data is all stored on a database server, so all we need is to reinstall the web interface and configure it.
So we look through the database for another server to take this interface. We find box21.example.com that can take the load.

We move the service SUGARCRM from the box01.example.com entry to the box21.example.com entry, and maybe move the alias `crm.example.com` to box21.example.com and commit the database change.
Then we push the change using:

```
sudo /config/Rdist -m box21.example.com
```

The instructions for establishing the SUGARCRM service:

- install the Sugar CRM package
- push the apache configuration file needed to access Sugar
- install the configuration file for the Sugar software (that connects to the database ...)
- do the other fiddly bits needed to make it work: change dns for `crm.example.com` to point to box21.example.com etc.

All of this is done by DACS. No out of date documentation or missed steps. Since these files were running on box01.example.com until its demise you have a high degree of confidence that they will work on the new system
You run the distribution command twice to make sure that all the updates are in place. Then you run the distribution command to push the dns changes. Maybe one more run to detect any other cascading changes from the service move. While the last distribution is running, you try `http://crm.example.com` and you log in using the administrator account and password and it looks like it's working.
Meantime you get a report from the hardware tech who says that the mother board failed and they will have to replace the system. So that will be a couple of hours or so at least.
Then you work on getting the TWiki development installation set up. So within 45 minutes you have replaced one critical piece of software and you are starting on the second.
Now how does this change without DACS?
Well the written documentation you have doesn't mention that Sugar CRM was deployed there. Indeed you spend a fair amount of time poking at the system so you can try to get the configuration info off the box.. Finally you transplant the disks to another system, but it's a different controller and can't make sense of the raid format on the disks. By this time 45 minutes has gone past.
Meanwhile, the help desk was notified that Sugar was dowe, and they found somebody else to work on it. After having gone to the box where Sugar was documented to be deployed (and taking 25 minutes to figure out that the remnants deployed there have not been operational for months), he is looking at DNS to try to figure out what box really provides crm.example.com. Ok, he finally finds it and goes to the system. He finds the hardware tech there who mentions that you are working on it.
So finally the two of you are working on gettng the prior night's backups restored somewhere so you can pull the files you need to set up the services somewhere else.
You do get things restored — finally.

2.3 Use case modifying a managed resolv.conf

This section discusses manually maintaining a resolv.conf file using a subversion based DACS installation.
The first thing to do is to check out the DACS tree. Usually this is done once when you get started with DACS and you maintain an up to date working using `svn update` from then on. However for now we do the initial checkout using a command that looks something like:

```
svn checkout svn+ssh://dacsuser@dacsmaster/repo/Config/work DACS
```

where the `svn+ssh...` says to connect using ssh to the machine `dacsmaster` using the user `dacsuser` and check out a copy of the `/Config/work` tree to a subdirectory called DACS in your current directory.
The next thing to do is to find the resolv.conf file in the DACS CCM tree. Looking at the directory listing of the DACS directory you see a directory called `dns`. Changing to `DACS/dns` you see the file resolv.conf.
Invoking an editor on it, you see it is a copy of /etc/resolv.conf so you feel confident that you have the right file. (You could also consult the files-report section 6.7.2 to get a mapping between the DACS work tree and the destination file/host.) You add a new `nameserver` line.
Now you commit the change to the file using `svn commit resolv.conf`. This will start an editor for you to add comments on what the change was made and which ticket/request needed this change. These comments form the basis of discovering why the change was made (the ticket information) and details about the change. In this case the following should suffice:

```
ticket:2345 brought up a new dns server for redundancy at the lax1
site. Adding it to the site-wide resolv.conf so systems can use it.
```

save the change and exit the editor finishing the commit.
Now you want to push that change to all your hosts. You do this using the `Rdist` command. A simple command line summary for the `Rdist` distribution script is:

```
sudo /config/Rdist label
```

which invokes the rules in the Distfile (the file that controls how files are distributed using Rdist) that have the label "label". Now how do you know what label to use? By convention the label name is the same as the name of the directory under the root of the CCM tree.
(Note: there can be multiple independent DACS CCM trees, by specifying /config/Rdist we specify that the tree under /config will be updated and used to manage the systems. Why do you want multiple trees? See "Using multiple DACS master trees" section 8.1. It won't be discussed further at this point.)
In this case you are in the dns directory, so you expect the label is dns. So you could run `sudo /config/Rdist dns` to push the changes under the dns directory to all the hosts. Since resolv.conf is under that directory it's change will be pushed.
But suppose you wanted to verify that only resolv.conf would be updated, since there are other files in the dns directory. Running `sudo /config/Rdist -S c -v dns` would give you a condensed summary (`-S` summary `c` condensed) of the changes.
After running the verify you see that the machine `a.example.com` was getting a zone file pushed as well. Well you don't want to update that, so you decide to not update `a.example.com` just yet.

Running `/config/Rdist -S c --exclude a.example.com dns` would push the dns label/directory to all your hosts except `a.example.com`.
If there are no errors or hosts down, that's pretty much it. If there are hosts down, a future verify run (using `-v`) will report that the file needs to be pushed and it can be done at that time. Also the future verify run (which is often run daily and sent as email to administrators) will show resolv.conf out of date on `a.example.com` so you don't have to remember that a host still needs an update. It will be automatically remembered for you.

2.4 Use case setting up to manage sudoers file

The prior use case was just pushing an already exiting file. It was already managed by DACS. Now we are going to look at establishing a file under management.
Things start like before by checking out a copy of the DACS CCM tree using `svn checkout`. Now we have to choose a subdirectory in which to store the file. We have a few choices:

- the file lives at /etc/sudoers, so somewhere under the `etc` directory makes sense.
- there is also a users directory with configuration files for particular users, and since sudoers configures users to do certain tasks with elevated privileges this may be a good spot.

Oh decisions decisions. The nice part is that you can rearrange it later if it seems that your placement isn't very good. So we decide to put it under `etc/sudoers/sudoers`. We make the new `etc/sudoers` directory (`etc` exists already) and create the sudoers file. Then we use subversion to add the new directory (and file), set the more restrictive permissions on the file when it is checked out, and commit the changes using:

```
svn add etc/sudoers
svn propset svn:unix-mode 400 etc/sudoers/sudoers
svn commit etc/sudoers
```

we add the comments for the commit as in prior use cases.
Now we have to define where to install the sudoers file we just committed. To do so we change to the directory `Config/distfile` under the DACS CCM root. Edit distfile.base and look for `etc:`, which is the way to specify the label `etc` in a distfile. Because we placed the file under etc/sudoers/sudoers, etc is the appropriate label for distributing this file. Once we find this section we add a couple of new distribution rules:

```
etc:
$C/etc/sudoers/sudoers -> ${ALL_HOSTS}
        SAVEINSTALL(/etc/sudoers, 3);

etc.sudoers:
etc:
$C/etc/sudoers/sudoers -> ${ALL_HOSTS}
        SAVEINSTALL(/etc/sudoers, 3);
```

Note that we have duplicated the same rule but with two different labels. In one case we used the label `etc` in the other we used the label `etc.sudoers`. Remember in the first use case we

excluded `a.example.com` from the update because we were pushing multiple files using the label `dns`? Well the `etc` directory has a lot of files in it and because sudoers is a security related file, delaying it's distribution is something to avoid. By creating the label `etc.sudoers`, we can push just the `sudoers` file without the rest of the files in the `etc` directory tree.
The `etc` target is used for automatic update and verification of the sudoers file. When Rdist is run without a label, the `etc` label is automatically selected and the `sudoers` file is automatically updated or verified on all managed hosts.
In both cases the rule is the same:

- it pushes the file located at `$C/etc/sudoers/sudoers`, (i.e. the file you just created under the CCM that Rdist is running from. `$C` stands for the current configuration root).
- to `(->)` some hosts. In this case is pushes it to the predefined class ALL_HOSTS which includes all the DACS managed hosts.
- SAVEINSTALL installs the file at `/etc/sudoers` on each host saving the 3 prior copies of the file.

Check-in `distfile.base` using `svn commit distfile.base` and add a log message.
Now when pushing the `sudoers` file you have two choices:
`sudo /config/Rdist -S c etc.sudoers`
or `sudo /config/Rdist -S c etc`.
While both will do the job, using the `etc` label may also push some other files, so run the first alternative.
`sudo /config/Rdist -S c etc.sudoers`
You are done adding a new file to DACS control and distributing it to all your hosts.

2.5 Use case set up to manage ntp.conf on all hosts from database

This is covered in detail in the Setting up a database driven ntp configuration section 7.5, but this introduces the basic steps.

2.5.1 Planning the network time protocol setup

NTP distributes time from server to server. A usual setup for a site it to have three machines on the network that talk to server external to the network. Then all the hosts on the network talk to all three of those servers.
In the database you specify:

- the services a host supplies section 3.2.2

and the

- services the host uses section 3.2.2

So we will define in the database three service values indicating that the service with that value is one of these externally talking servers. The service names will be `NTP1`, `NTP2`, `NTP3`. Also we will create a uses value that indicates that a host should talk to these three servers. The uses value will be NTPCLIENT.
Now that the structure is established, we edit the dbreport script to define the service values NTP1, NTP2 and NTP3 as well as the uses value NTPCLIENT. In the DACS database we assign the service NTP1 to one host, NTP2 to another host and NTP3 to a third host. Then we assign NTPCLIENT to all the machines that need time synchronization (excluding the three NTPx servers).
Once this is done:

1. create a new directory ntp
2. in the `ntp` directory create a subdirectory `dist` that will hold the generated files that are to be distributed
3. create a Makefile that will query the database (using dbreport) and extract the IP addresses of the three NTPx servers to include in an ntp.conf file that is sent to the NTPCLIENT machines. This file will be located in dist/clients.conf.
4. manually create the files ntp_root.conf that will be used to generate the files: dist/ntp_root1.conf, dist/ntp_root2.conf, dist/ntp_root3.conf to be pushed to the three top level servers.
5. modify Config/distfile/distfile.base to push the three dist/ntp_rootX.conf files to the hosts in the NTP1_HOSTS, NTP2_HOSTS and NTP3_HOSTS classes. (Note: all services values generate classes ending in _HOSTS.)
6. modify Config/distfile/distfile.base to push the dist/clients.conf file generated by the Makefile to the hosts in the NTPCLIENT_SLAVES class. (Note: all uses values generate classes ending in _SLAVES.)

Check in the changes and push using `sudo /config/Rdist ntp`. Now this seems like a lot of work, and it is. This is just the basic layout and leaves out a lot of steps (which are covered in the DacsExamples section).
But what does it gain us? Well if you move the NTP1 service from one host to another it keeps all the clients (and the other NTPx servers) in sync automatically. You never have to worry about editing the ntp.conf file that is distributed because you moved an NTP server. The Makefile will discover that the database has changed and it will regenerate the list of NTPx IP addresses and then rebuild all the ntp configuration files using the new information.
How many times have you logged into a system and found out that the configuration of some service no longer matched reality? It was correct when deployed, but over time it became incorrect.
Automatic mechanisms like this prevent the configurations from diverging. They can also deploy other changes such as modify firewall configurations (to allow ntp traffic to/from the new ntp servers) and eliminates the manual steps required to keep all the info in sync.
Whether your site needs this level of automation is up to you to decide. Simply having all the files managed manually may be enough as they are all in one location, so when you change the NTP

configuration you can manually search all the files in the DACS CCM for the old IP address (or hostname) and change it. Imaging trying to do that if all the files exist only on the end machines:

- login to each machine and a manually edit the file
- run some remote command from a master machine that loops over all the hosts. Have the command edit the configuration on each machine. You hope the command script is robust enough to not get confused on some variant of the config file that leaves the system broken.

I am sure you can come up with more scenarios, but these sorts of problem are what any CCM system is designed to handle.

Chapter 3

The "database" system

There are two components to the system:

- the database file
- the db_report command that verifies and quieries the database

3.1 What does the database provide

The database is intended to be the authoritative inventory of information for all systems. By extracting this information from the database and including it in configuration files, you prevent having to specify and later update the same information in multiple locations. Having all the information in the database also makes obtaining a consistant view of your system configuration much easier.
Each host under DACS control *MUST* have an entry in the database. You can also inventory hosts not under DACS management in the database. Then you can use it to quickly identify a machine with a particular Ethernet address section 3.4.2.6, or generate a report of all your assets section 3.6.1.

3.2 The database file

The term "database" is somewhat of a misnomer. It is not a SQL database or even a binary file. It is one or more hand editable text file(s) using a `keyword=value` syntax.
The database file is a simple text file. All database entries start with the keyword `machine` and continue to the next `machine` keyword or end of file. The value of the `machine` keyword *MUST* be the same as returned by the hostname command run on that host. Duplicate machine names are not allowed in the database. Hence using a FQDN (fully qualified domain name) or other unique name for you hosts is important. A sample entry is:

```
machine = host1.example.com   (*)
alias =  host1
arch =  i686
cluster = site_lax1 host_dev host_qa (*)
comment = this is an old machine and should go away
```

```
contact = steve@example.com
cpu = intel
disk = WD2500AB(250GB) WD5000YS(500GB)
enet = 00:00:34:02:00:ab
enet_card = forcedeth
enet_if = eth0
hostid = 667154
hub = switch1
hub_port = 10
ip = 192.168.4.3/24 (*)
location = LAX rack 105.4 U 23-26
maint_info = support contract A40560 till 12/01/2007
model = AS1445
os = CENTOS 5.2 (*)
patches = forecedeth
pmemory = 16G
purpose = generic dev/qa box
rdist = yes (*)
services = APACHE TOMCAT (*)
services = SSHD (*)
sn = 667154
tag = A0112
uses = 3WARE NTP (*)
wall_plug = cable  5
```

The *'ed items above are required. A minimal configuration entry for the same host would be:

```
machine = host1.example.com
cluster = site_lax1 host_dev host_qa
ip = 192.168.4.3/24
os = CENTOS 5.2
rdist = yes
services = APACHE TOMCAT
services = SSHD
uses = 3WARE NTP
```

This entry would provide basic functionality. However when generating a wiring report section 3.6.3, the host would be missing since there is no wiring information in the minimal entry. Also, if you wanted to find all your hosts with 16GB or more of memory, host1 wouldn't show up if entered using the minimal configuration. Fortunately it would show up if you were searching for hosts that are missing memory info.
The services entry occurs multiple times. Declaring certain keywords multiple times concatenates the contents. So in this case I could also have specified:

```
services = APACHE SSHD TOMCAT
```

however if certain groups of services are deployed together (like Apache when it acts as a front end to tomcat), they can be grouped together on one line in the specification to make the dependency more obvious and make deleting the configuration easier.
I have been asked why DACS doesn't use a real database like postgres or mysql. Well there are a few reasons:

- DACS should operate with minimal resources. It is possible to override most of the safety checks and distribute files even if things are very broken.
- For redundancy you can have multiple DACS master hosts. Each must be able to work independently. Using a real database would require multiple synchronized db setups and further raises the requirements.
- How do you version control a database? If somebody makes and commits a mistake in the database, how do you get back the prior configuration? The mysql database files aren't under version control, so what mechanism would allow a rollback to a known good configuration?
- Some file based mechanism like SQLite may work, but there is still the problem of defining a normalized schema.

At least one site did use a SQL database back end, and they found that they were doing a lot of multiple joins to extract the info they needed for build/deployment, and it was taking a long time. So they used the database as a front end to restrict who can change host configurations. Then they exported dbreport (at the time class_gen) compatible files on any change. These exported files were checked in and the database could be reloaded from the files to allow rollback. The update time on the exported file was used to trigger builds when the data changed.
As mentioned above, the database has a number of keywords which are defined in the following section.

3.2.1 Keyword definitions

A full description of each database field is given below: The following table describes the keywords. In the table are:

- the keyword name
- whether the keyword can be used multiple times. If so then the values are concatenated.
- the format of the value
- a description of when it's used

Table 3.1: Keyword fields defined for use in the DACS database

keyword	**multiple**	**format**	**description**
alias	yes	space separated aliases as in /etc/hosts	hostname aliases
arch (3.3.1.4)	no	text	base architecture (i586, microsparc)
base	no	text	set to machine name of complete host. If not specified, set to the "machine" name. See below for further info.

cluster (3.3.1.1)	yes	enumerated	What group is the host in. Can be site, device role (production development, test)... defined in dbreport.
comment	yes	text	comments about machine
contact	yes	text or email address	space separated set of email addresses (main users of machine)
cpu	yes	text	type of cpu (sparc, intel, mips)
disk	yes	separate list of disk types/sizes	disks space
domain	no	text	domain of system
enet	no	: separated lower case hex characters 0a:ff:ec:32:df:ff	the interfaces Ethernet address in lowercase
enet_card	no	text	the network interface type, 3com...
enet_if	no	native form of Ethernet interface on system	Ethernet interface identifier. E.G. eth0, eth1.1, eth0:1, nge0, lo:2
fqdn	no	FQDN	fully qualified domain name automatically set by machine keyword. Name of field used for -f fqdn output.
hostid	no	text	software accessible unique identifier
hub	yes	machine name for hub	hub/switch for device
hub_port	yes	text usually number	port on hub
ip	no	text	ip address in CIDR notation w/ optional network prefix
karch (3.3.1.4)	no	text	kernel architecture (a cpu/arch mix) used on Solaris for example sun4 is the base arch, but sun4m is the cpu. For pc based this could be amd64 where the base is x86_64
location	no	text, usually site, rack, numbered shelf/U measurement	location of the computer
monitored	no	monitored service list	text list/description of monitoring of machine
machine	no	text	the FQDN, or base name of the machine
maint_info	no	text	maintenance info, contracts, contact info etc.
model	no	text	manufacturer model identifier
os (3.3.1.4)	no	enumerated value and version string	the name and version of the operating system.
patches	yes	text with comma and space between patch identifiers	list of patches to apply

pmemory	no	number suffix (M,G)	amount of physical memory
purpose	yes	text	free form text of purpose of machine
rdist	no	yes/other/request/no	value yes if under normal rdist maintenance, request if only manual specification of host should push files, no if not to be pushed by rdist. Other for alternate file distribution method . Default no.
services (3.3.1.2)	yes	enumerated ", space" separated list	the services a host provides. (e.g. runs LDAP server)
sn	no	text	serial number of device
tag	no	text	asset tag number
uses (3.3.1.3)	yes	enumerated ", space" separated list	the services a host uses or local configuration info (e.g. configured to use LDAP servers)
wall_plug	no	text	identifier for wall plug device is connected to, or cable label for direct patch connect.

Internally the following keywords are defined and can be used for searching, but are not specified in the configuration.

subnetmask specified with the `ip` keyword by using a CIDR or netmask format. For example 1.2.3.4/24. Be consistent and choose only one way of specifying the mask. Otherwise searching on the netmask will be a pain. No conversion between forms (CIDR/netmask) is done.

isbase set to yes if the machine definition does not include the `base` keyword. Set to `no` otherwise. Note that the base keyword is always set section 3.2.3 even if `base` is not specified.

3.2.2 Further comments on the service, users, cluster and os keywords

The file Config/database/db section 3 file has 4 keywords that are used to put hosts into classes. The keywords correspond to each class type section 3.3 and are:

services section 3.3.1.2 the list of services provided by this host. Usually involved running a daemon of some sort.

uses section 3.3.1.3 the list of services that a host uses or local configuration information about the system.

cluster section 3.3.1.1 Other info about a host such as site info, system role (production vs. development vs. test). Basically anything you want to classify that doesn't fit into the uses/service or os keywords.

os section 3.3.1.4 The os and version of the OS running on this machine.

To decide what keyword a particular configuration option should use ask the following questions:

1. If the system with this configuration value dies, will you need to replicate the configuration bound to the token on another host?

2. Is this something that describes the system hardware or software?

3. Is this an administrative or other external to system configuration item (e.g. location)?

If you answered yes to question 1 then it's a `services` value. If you answered yes to question 2, then it's a `uses` value. If you answered yes to question 3 then it's a `cluster` value. If you answered no to all of them, well it's a cluster value and I would like to know what it is as it's a new use case.
Examples:

- The configuration of an LDAP or NIS server is a service. You will need to bring up a replacement server/service if the host running it dies.

- If a host needs a particular script run to set up autoverify on its 3ware card, then the token identifying the system as running a 3ware controller belongs to the uses keyword.

- If the machine is located in Los Angeles and is used by development, these tokens are part of the cluster keyword.

These aren't hard and fast rules, e.g. SSHD is defined as a service which probably doesn't matter if the host goes down, but ssh is used as transport/access method for other services (e.g. DACS management) and any replacement box will probably have to run it as well so it is probably best defined as a service.
Uses and service tokens consist of capital letters, numbers and underscores '_'. The cluster keyword adds lower case letters. This is a historic artifact that may be relaxed in the future allowing mixed case for uses and services.
Note that underscores '_' in uses and services have a special meaning. See the Database output/Services section 3.3.1.2 section below.

3.2.3 Inheriting information from another host (creating child host)

Using the `base` keyword allows a new machine entry to be based on a prior entry. The following entry for an alternate interface uses the `base` keyword:

```
Machine = tiger-if2.example.com
base = tiger.example.com
enet = 00:07:e9:e6:e4:85
hub = sc02_mht1
hub_port = 15
ip = 138.84.130.136/26
rdist = no
wall_plug = (cable unlabeled)
```

This looks at the (preceding) entry for `machine=tiger.example.com` and copies/inherits any undefined values from the base machine. The only keyword not copied is `rdist` since there is (generally) no sense in distributing to a given machine more than once, and the default value for `rdist` is no. (Note that you can explicitly specify `rdist=request` in the entry to make rdist use this machine name if you need to distribute using both machine names).
The `base` keyword is always set to the value of the machine entry where a base value is not set. For example:

```
machine = a.example.com
...
[no base keyword is used]

machine = a-if1.example.com
...
base = a.example.com
...

machine = a-if1-1.example.com
...
base = a.if1.example.com
```

The value of base for both `a-if1-1.example.com` and `a-if1.example.com` would be `a.example.com` since the machine entry for `a.example.com` does not specify a base keyword.
The entry referenced by the `base` keyword is expected to be the host that is used for rdist access, if the system is maintained by rdist. There is nothing that enforces this requirement on base (i.e. the base need not be an rdist host), but it is a useful convention.
The `base` keyword is evaluated when a host is seen in the config file. Thus the machine referenced by `base` must be defined before the reference (this also eliminates the possibility of circular references).
Any database entry that does not have an explicit `base` keyword has the `base` keyword set to the value of `machine`. This makes it easy to select the base host and all of the systems derived from it. If you want to select a host that is not a base host (i.e. that has the base parameter):

- use the dbreport option `-s 'isbase:/no/'`

You can do the same operation differently by not selecting a host without the base parameter:

- use the dbreport option `-n 'isbase:/yes/'`

See the dbreport documentation section 3.4 for more information and examples.
For example, suppose you have a host `ftp.example.com` that can be moved between systems. Currently it is assigned to `server.example.com`. If the base keyword for `ftp.example.com` points to the rdist identity of the actual machine, then

```
dbreport -f base -s 'machine:/ftp.example.com/'
```

will generate `server.example.com` the name of the host that will get the files to install/activate `ftp.example.com`. The dbreport command line

```
dbreport -f machine -s 'machine:/server.example.com/|isbase:/no/'
```

will return the list of machines that have `server.example.com` as their base but it will not return `server.example.com` itself.

3.2.4 Representing multiple IP addresses for a host

A DACS database entry only allows for one network interface. To support multiple network connections on a single host, use the base keyword to link a new host entry with just IP/Ethernet info to its parent entry. A sample entry looks like:

```
Machine = tiger-if1.example.com
base = tiger.example.com
enet = 00:07:e9:e6:e3:85
hub = sc02
hub_port = 15
ip = 227.136.10.23/26
rdist = no
wall_plug = (cable unlabeled)
```

tiger-if1.example.com will inherit the unspecified values (such as os, services, uses ...) from the entry for tiger.example.com which MUST be defined before this entry is seen in the database. You can optionally use `services=NONE` and `uses=NONE` if you want to prevent the new host from inheriting the services or uses list from the base host.

3.2.5 Using a unique IP address for a service (e.g. DNS)

In many networks certain well known services are bound to a specific IP address. For example, the DNS server should always be at the x.x.x.23 address on the network.
To set this up use two machine entries. The first machine entry is used by the rdist system to push needed files. The second entry is used to identify the specific IP (and other info) that needs to be associated with the service.
For example, the machine 'server.example.com' runs a DNS server bound to the IP of the machine 'dns.example.com'.
To set this up we:

1. Define the service DNS in dbreport
2. Add the service (e.g. `services=SSHD ... DNS ...`) to 'server.example.com', which will be the base host for 'dns'. 'server.example.com' will have 'rdist=yes' set in its database entry.
3. Define a new machine with a new name (e.g. 'dns.example.com') and set its `ip` parameter to the IP address for the service (e.g. 192.168.1.23). Set its base parameter to the base host (e.g. `base=server.example.com`) and set the single service it provides (e.g. `services=DNS`).

You may want to extract the IP address for the 'dns' machine. We will discuss two ways to do it. They may be other ways to do it depending on how the database is set up. Here are two ways.
Use dbreport to select hosts that have only the DNS service defined:
`dbreport -f ip -s "services:/^DNS$/"`. The regular expression matches DNS and only DNS. So it will not match the `service.example.com` entry that probably include SSHD and other services. Using a slightly more complex dbreport command:
`dbreport -f ip -s 'services:/^DNS$/|isbase:/no/'` will work even if the base host `server.example.com` has only the single DNS service on it.

A second way to do this is to generate a list of all hosts that are being distributed to (`rdist=yes`) that have the DNS service (in this example that is only 'service.example.com', but in general may be multiple hosts) using:

```
dbreport -l -s 'services:/DNS/|rdist:/yes/'
```

For each host in that list, report the IP address using the base parameter to match all instances of the given host.

```
for host in `dbreport -l -s 'services:/DNS/|rdist:/yes/'`
do
  dbreport -f ip -s "base:/$host/|services:/DNS/' -n 'isbase:/yes/'
done
```

this command will return the IP address associated with the DNS server that is not a base server (i.e. dns.example.com). Note this second way fails if there are multiple child entries for the base host that inherit the services list. To prevent this explicitly assign the `services` keyword for the child entries. For example using `services=NONE` will solve the problem.

3.3 Database output (class definitions)

The output from the database is done by dbreport. Dbreports default output mode generates a list of rdist class variable definitions for all the selected hosts. The build system produces an identical list of class definitions for use by filepp. These classes can be combined using filepp and rdist commands to implement the standard operations on sets including:

- intersection
- difference
- union

To select a host that runs Solaris and is in Los Angeles, intersect the two classes LAX1_C_HOSTS and SOLARIS_HOSTS. The mechanics of this will be deferred to later (in Dacs distribution section 6.4.4 and DACS Build/Using filepp section 5.6.3), but this provides a powerful way of selecting hosts that receive specific configurations.
One thing to look out for is not to combine separate items in a single keyword. For example, don't define a service APACHECENTOS5 that describes a Centos 5 host that runs Apache, instead intersect the CENTOS5 and APACHE classes of hosts. This simplifies configuration as it reduces the number of unique services to select from as well as improving database accuracy as some items (such as OS type) are verified by the DACS system and you are notified if they are incorrectly set.

3.3.1 Class Types

There are 4 types of classes:

1. clusters section 3.3.1.1 - groupings by location, device purpose (e.g. network, external)

2. services section 3.3.1.2 - configuration of the box that is meant to be used by other software/systems or users.

3. uses section 3.3.1.3 - a *local* configuration option for the system, or a service that is used by the host.

4. operating system section 3.3.1.4 - classes per operating system and version and optionally arch and kernel arch.

(Note: there is also a patches class, but it is not used much although it will continue to be supported. Understanding the classes above should allow you to use the patches class if you find a need. If you do use the patches class, please let me know.)
Note that the services and uses classes use the underscore character to provide a very simple class inheritance like feature that is discussed later. None of the other classes do this.
The values for the classes are defined in an associative array in the dbreport script. See Config/bin/dbreport for the list of service, uses, os and cluster values that are defined for your site. If your site uses robodoc and generates documentation from DACS, you may have a document to look at rather than reading the dbreport script.
If you need to add/change the classes, see Changing dbreport section 3.4.3.

3.3.1.1 Clusters

The `cluster_list` associative array contains the names of all the valid items for the cluster keyword. If a host is put in the cluster, it is added to the class `X_C_HOSTS` (and filepp macro =FILEPP_X_C_HOSTS=) where `X` is the key (where case is preserved) in the `cluster_list`.

3.3.1.2 Services

The `service_list` associative array contains the names of all the valid items for the services keyword. When a service is added to a host, it is added to the `X_HOSTS` class and corresponding `FILEPP_X_HOSTS` filepp macro.
If the token includes an underscore, e.g. FIREWALL_MANUAL the host is added to multiple classes including: FIREWALL_HOSTS and FIREWALL_MANUAL_HOSTS for rdist and the corresponding filepp classes.
This is an attempt to support variations within a service definition. So FIREWALL_MANUAL can get all the settings of FIREWALL and the addition settings associated with FIREWALL_MANUAL. Note that only the last component is removed, so FIREWALL_MANUAL_FOO would add to the classes FIREWALL_MANUAL_HOSTS and FIREWALL_MANUAL_FOO_HOSTS but not FIREWALL_HOSTS. This is an area that is targeted for further improvement.

3.3.1.3 Uses

The `uses_list` associative array contains the names of all the valid items for the uses keyword. When a use is added to a host, it is added to the `X_SLAVES` class and corresponding `FILEPP_X_SLAVES` filepp macro.

As with services, using an underscore adds the hosts to multiple classes: e.g. DNS_FULL uses keyword will add the host to the classes DNS_SLAVES and DNS_FULL_SLAVES for rdist and the corresponding filepp classes.
This is an attempt to support variations within a uses definition. So DNS_FULL can get all the settings of DNS and the addition settings associated with DNS_FULL. Note that only the last component is removed, so DNS_FULL_FOO would add to the classes DNS_FULL_SLAVES and DNS_FULL_FOO_SLAVES but not DNS_SLAVES. This is an area that is targeted for further improvement.

3.3.1.4 OS/Arch/Karch

The `os_type_list` associative array contains the names of all the valid os types for the os keyword. It can also includes os declarations like "SMC" which is the "operating system" for an SMC switch. When the os keyword is defined for a host you specify a version for the os as well as the type. Then a series of classes are created. E.G. if the OS is `centos 4.3` the host is part of the following classes and corresponding filepp (named by prepending `FILEPP_`) macros:

```
CENTOS_4.3
CENTOS_4.X
CENTOS_HOSTS
```

Basically a macro is created for the os type and for each component of the version string. So to get a list of all CENTOS hosts running any version 4 of Centos, you would use the CENTOS_4.X class. The CENTOS_HOSTS class would have Centos version 4, 5, 3 etc. hosts in it.
If `arch` or `karch` is defined for a host you get some supplementary OS classes. There are two versions of arch support. Legacy (where arch is separated from the OS name by a dash. E.G. CENTOS-x86_64 or SINIX-D) and normal where the `arch` keyword is explicitly defined. I suggest you stick to the normal form.
If the arch keyword is specified and the OS includes a legacy arch (e.g. CENTOS-x86_64) a warning is generated and the OS supplied arch is used. This is a legacy support mechanism that requires disabling the arch validation mechanism in HOSTVERIFY. If you are converting from DACS, it is recommended that you remove the legacy arch OS types and use the arch keyword instead.
The arches are combined with the operating system classes since the usual use of these classes is to push files to a specific arch running a specific OS. Arch is usually the output of "uname -m" which is used by HOSTVERIFY to verify the arch of the host in the database.
Sample classes:

```
CENTOS_Ax86_64_5.1 - 64 bit hosts running centos 5.1

CENTOS_Ax86_64_5.2 - 64 bit hosts running centos 5.2

CENTOS_Ax86_64_5.X - 64 bit hosts running any version of centos 5

CENTOS_Ax86_64_HOSTS - 64 bit hosts running any version of centos

FEDORA_Ai686_2.0 - 32 bit host running fedora 2.0

SOLARIS_Ai86pc_5.10 - Solaris box running Solaris 5.10 (I assume 64 bit)
```

It is an error to define a host in the database without an os value, so a host will always exist in some set of OS classes. A host will be listed in an arch class *only if* the arch is defined. So a host running Centos 5.1 on an x86_64 platform will be in the classes:

```
CENTOS_5.1 - all hosts running Centos 5.1
CENTOS_5.X - all hosts running any version of Centos 5
CENTOS_HOSTS - all hosts running any version of Centos

CENTOS_Ax86_64_5.1 - 64 bit hosts running Centos 5.1
CENTOS_Ax86_64_5.X - 64 bit hosts running any version of Centos 5
CENTOS_Ax86_64_HOSTS - 64 bit hosts running any version of Centos
```

But a host without an arch value will be in only the first three classes.
Note there is no class of all x86_64 hosts. In past experience it isn't very useful as these classes are used for kernel modules and other lower level stuff that is usually tied to both the OS and the system or kernel architecture.
Speaking of kernel architecture similar classes are defined:

```
CENTOS_Kx86_64.opteron_5.1 - 64 bit hosts with karch of x86_64.opteron
                             (i.e. opteron processor) running centos 5.1
CENTOS_Ki386.xeon_5.X - 32 bit hosts running xeon processor
```

The arch and karch identifiers are very flexible. Neither is coded in dbreport like an os or services value. The data verification occurs in HOSTVERIFY so verifcation can be a more complex procedure if needed. This can be changed if it seems warranted.
The karch id is not verified by HOSTVERIFY at this point. If you want this, I recommend modifying HOSTVERIFY to use something like
‘uname -m‘.processor_name
on Linux and just ‘uname -m‘ for Solaris (if you are using a sparc chip, uname -m returns things like sun4m, set the arch to sun4 when the karch is sun4m, sun4u etc).

3.4 dbreport

Dbreport is a Perl script that queries a simple `keyword=value` style database where each new entry starts with the `machine` keyword. It also implements the schema and validates data entered in the database.

3.4.1 The command line

```
/config/Config/bin/dbreport: [-l | h | r | f field[:field]]
         [-d "string"] [-D "string"] [-GH]
         [-F "format"] [-[sn] <field>:/pattern/[|field:/pattern/]* ]
         [database_file(s)]

 Output type selectors
  r - rdist group style output (default)
  l - list of host names
  h - /etc/hosts style output.
```

```
 f - output the values for a list of colon separated fields.


Output format selectors with -f flag.
 d - delimiter between fields in f style output
 D - delimiter at end of line for f style output
 F - format string with %f replaced by field name and %v replaced by value.
               E.G. '<tr><th>%f</th><td>%k</td></tr>' will create a single row in
               a HTML table.
 H - make first line a header line

Misc
 G - enable debugging output

Machine selectors
 n - select machines with field(s) not matching pattern(s).
 s - select machines with field(s) matching pattern(s).

Patterns are Perl regular expressions. A | (alternation) can be used
in a regexp provided it is not preceded with a /. If you need to
match /| in a pattern use [/]| instead.  Multiple field:/pattern/
pairs are and'ed together so the machine MUST have all patterns pass.

Use -? to get this help.
```

There are many example command lines scattered throughout the documentation and there is a computer based training module in the file CbtDacsDbreport.txt that will lead you though the basics. More examples are present in the report shell scripts in Config/database/reports. You can also read the examples below.

3.4.2 Examples of dbreport usage

Dbreport takes a number of arguments, by default, it will use /config/Config/database/db on the local machine as its database. If you use it on a system that is not a DACS master, you need to supply the database file name.

3.4.2.1 Query selection tips

The -n and -s options are used to exclude/include (Not select/Select) host records from reporting if their criteria match.
Both take a series of `<keyword>:/<pattern>/` sequences separated with a `|`. So to select hosts that are at the first Los Angeles site (in cluster site_lax1) and runs any version of Centos 5, you specify:

```
-s 'os:/(?i)centos\s+5/|cluster:/site_lax1/'
```

All the `|` separated portions sequence must match. So the host must match both the `os` and the `cluster` selectors. (Arguably rather then using `|`, `&` should have been chosen to separate the elements indicating that they are and'ed together but....)
Because of this and'ing, you can't select hosts in Los Angeles and Miami using:

```
-s 'cluster:/site_lax1/|cluster:/site_mia1/'
```

because a host can't be in *both* Miami and Los Angeles (quantum mechanics not withstanding). This is done using the appropriate regular expression:

```
-s 'cluster:/site_lax1|site_mia1/'
```

so cluster matches `site_lax1` or `site_mia1` and returns the desired result.
Want to select all Centos and Solaris hosts? Perform a case insensitive match by starting the regular expressions with =(?i)=:

```
-s 'os:/(?i)centos|solaris/'
```

Want all Centos and Solaris hosts excluding Solaris 10 hosts? This works by selecting all hosts and excluding the Solaris 10 hosts:

```
-s 'os:/(?i)centos|solaris/' -n 'os:/(?i)solaris\s+10/'
```

This creates the same output (assuming you only have solaris 9 and 10 hosts) by explicitly selecting only Solaris 9 hosts:

```
-s 'os:/(?i)centos|solaris\s+9\./'
```

Note that you can't search for an alternative string ending in a / because:

```
ip:/2.21/|2.220/
```

is mistakenly parsed as the `/|` character string that is meant to separate alternate `<keyword>:/<pattern>/` sequences, and not as find: '2.21/' or '2.220'. So use:

```
ip:/2.21[/]|2.220/
```

instead.

3.4.2.2 Get a list of all machines in database

To generate a space seperated list of all machines in the database:

- `/config/Config/bin/dbreport -l`

```
a.example.com androscoggin.example.com b.lax1.example.com ...
```

3.4.2.3 Get a list of all our machines in database

There are a couple of ways of doing this:
Select all machines that contain .example.com assuming you are using FQDN's for the machine keyword:

- `/config/Config/bin/dbreport -l -s 'machine:/\.example\.com$/'`

this may miss some equipment like unmanaged hubs if they aren't assigned a (dummy) FQDN dns name.
Another way is to assign external equipment to the external cluster and run:

- `/config/Config/bin/dbreport -l -n 'cluster:/external/'`

to report all equipment without any equipment in the external cluster.
These commands also include machines that inherit from other machines. If you want just base machines you can add/use the selector `isbase:/yes/` as well.

3.4.2.4 List hosts with a particular OS

Find all Centos 4.5 hosts by searching the os keyword for a case insensitive "centos" separated by space from "4.5":

- `/config/Config/bin/dbreport -l -s 'os:/(?i)centos\s+4\.5/'`

produces:

```
dns.lax1.example.com s2-e1.lax1.example.com s2.lax1.example.com
```

3.4.2.5 Find all entries/interfaces/IP addresses for a given host

To find all IP addresses for the host known as s1.example.com use:

- `/config/Config/bin/dbreport -f ip -s 'base:/s1.lax1.example.com/'`

which returns:

```
  192.168.2.1
  192.168.2.23
  192.0.2.72
```

if you want the IP address followed by a space and the machine use:

```
 /config/Config/bin/dbreport -f ip:machine -d" " -s 'base:/s1.lax1.example.com/'</verbatim
to produce:
{\small \begin{verbatim}
  192.168.2.1 s1.lax1.example.com
  192.168.2.23 ldap.lax1.example.com
  192.0.2.72 ftp.example.com
```

Also you can use the -h option to get hosts style output with aliases:

- /config/Config/bin/dbreport -h -s 'base:/s1.lax1.example.com/'

to get:

```
  192.0.2.72    ftp.example.com         s1
  192.168.2.23     ldap.lax1.example.com   ldap.lax1
  192.168.2.1      s1.lax1.example.com     s1
```

3.4.2.6 Finding a host with a given Ethernet or ip address

Use the search capabilities of dbreport to list a host with a specific IP address:

- `/config/Config/bin/dbreport -l -s 'ip:/192.0.2.72/'`

to get `ftp.example.com`. To search by Ethernet address:

- `/config/Config/bin/dbreport -l -s 'enet:/^00:30:84/'`

you get `b.lax1.example.com ftp.example.com`. We can also select with a full Ethernet address, but searching using the prefix can tell us how many cards we have from a given manufacturer as well.

3.4.2.7 Getting only the primary machine name

When selecting by os, we see duplicates:

```
% /config/Config/bin/dbreport -f machine -s 'os:/(?i)centos\s+5\.3/'
s2.lax1.example.com
dns.lax1.example.com
s2-e1.lax1.example.com
```

This makes sense since many of the hostnames are the same host so if one is running Centos 5.3 the others must as well. However we often want to de-duplicate this. To do this we can rely on the convention of setting the "rdist" keyword to "yes" only in the primary machine. So we simply put:

```
rdist:/yes/
```

in a `-s` statement

```
%/config/Config/bin/dbreport -l -s 'os:/(?i)centos\s+5\.3/|rdist:/yes/'
s2.lax1.example.com
```

or use `-n 'rdist:/no/'`.

```
% /config/Config/bin/dbreport -l -s 'os:/(?i)centos\s+5\.3/' -n "rdist:/no/"
s2.lax1.example.com
```

to select the one true name/entry for a host.
This sometimes does not work (e.g. there may be more than one entry for a host with the `rdist` value set to `yes`). In this case use: `-s 'isbase:/yes/'` or `-n 'isbase:/no/'` to select the base entry for the host.

```
% /config/Config/bin/dbreport -l -s 'os:/(?i)centos\s+5\.3/|isbase:/yes/'
s2.lax1.example.com
% /config/Config/bin/dbreport -l -s 'os:/(?i)centos\s+5\.3/' -n "isbase:/no/"
s2.lax1.example.com
```

3.4.3 Changing dbreport

Often there is a class or keyword already defined that you can use or combine with other classes (using set operations) to define the proper set of hosts. However when there isn't you need to add a new class.
Dbreport not only generates reports from the 'database' it also implements the schema and validates the data. So adding new fields, changing enumerations like services or uses etc. requires changing the Perl script.
One question I often get is why not have an external configuration file(s) for the schema. Dbreport is run many times during a typical DACS push and I don't believe I can implement a faster parser for the schema than what Perl provides. So implementing an external config file would slow down the execution of dbreport. One possibility is to extract the schema defining components to an external library file and use the Perl library mechanism to include and parse it. Then the external file can be edited rather than editing the whole script. I haven't gotten around to testing that and don't see that it would be dramatically better than editing the dbreport file itself.
There are 2 types of changes you will typically do in dbreport:

1. changes to the clusters, services, uses, patches or os value lists

2. changes to the keywords defined in the database

but before we discuss making changes, you need to decide what changes you want to make. Usually deciding what service to add is easy, however when running multiple redundant servers or master/slave services a little thought it needed. See Special Circumstances section 3.5 for a discussion and references to how to set up the database for linked services.

3.4.3.1 Adding/changing cluster, services etc

When you change the cluster, service, os, or uses values you modifying a Perl associative array. The array names for each keyword are:

Keyword	*Associative Array*
cluster	cluster_list
os	os_type_list
patches	patch_list
service	service_list
uses	uses_list

Table 3.2: The associative array that defines a keword's values

To change these edit Config/bin/dbreport and search for '(%[arrayname])'. E.G. to find the start of the service_list array you would search for `(%service_list)`. Once you have found that the syntax is very easy. A sample entry looks like:

```
#****d* service_list/SSHDEXT
#* NAME
#* SSHDEXT - Runs ssh server with external interface
#* DESCRIPTION
#* Runs the ssh server and allows access from the Internet. Modifies
#* services list, firewall rules and ssh configurations.
#******
    'SSHDEXT' =>  '',
```

There is only one active line here `'SSHDEXT' =>  '',`, all the rest start with a '#' sign and a are Perl comments. This defines a new service called SSHDEXT. You can add new lines similar to the `SSHDEXT` definition and change `SSHDEXT` to some other value. Add the new line after any comma in the list.
The comment lines are robodoc documentation lines that describe what the service keyword (SSHDEXT in this case) is to be used for. These comments can be extracted and turned into HTML or printable documentation for use by others, and you are strongly encouraged to document your additions. See the robodoc manual at:
`http://www.xs4all.nl/~rfsber/Robo/manuals.html`.
The same format works for all the other arrays with the exception of the robodoc strings (the services_list turns into uses_list for uses for example). There are example entries for each of the arrays, and you can use those as templates for your robodoc comments.

To delete a keyword, simply delete the active line and its documentation. There are a helper programs in Config/bin that can be used to search for old values so you can clean up your files. Use `Config/bin/cleangrep` to look for instances of the keyword in just the subversion controlled files.

3.4.3.2 Adding a new keyword to the schema or change its verifier pattern

To add a new keyword to the schema chane the keyword_list associative array. An entry looks like:

```
'alias' => '0',
```

and is composed of a key and value. In this case the key is 'alias' and the value is '0'. If the value is 1, the keyword can be used only once per host definition in the database. If the value is 0, it can be used multiple times and the values are concatenated.
You can also define a verification mask that restricts the value of the data that is entered for this keyword. The mask is a Perl regular expression. If you aren't familiar with them there are numerous tutorials on the Internet. The `ip` keyword uses a mask to catch simple typos, its definition is:

```
'ip' => '1|[0-9.X]+(/[0-9]{1,2})?',
```

the parts of the value are separated with a `|`. The initial 1 means that `ip` is a single use only keyword. Then comes the validation expression(s). Every `|` separated regular expression is matched against the value of the keyword and if any of them pass, the value is validated. In this case there is only one validation expression that is explained as:

[0-9.X +] Allow any sequence with one or more digits 0-9, periods and the letter X. So an ip could be 192.168.X.X, or 173.34.56.78. The regular expression could be better written as it also allows 123456789 and 999.999.999.999 which are obviously invalid but it catches most typos.

(/[0-9 1,2)?] Allow 0 or 1 copies (i.e. it's optional) of a '/' followed by 1 or 2 digits. Again this allows /44 which is invalid, but it catches most cases.

Using multiple `|` separated regular expressions additional valid values can be specified. E.G. to validate enumerated keywords we have:

```
'rdist' =>  '1|yes|other|request|no'
```

which allows only the 4 values: yes, other, request and no.
Similarly to add the 'eri' Ethernet interface name, change:

```
'enet_if' => '1|eth[0-9](?:[:.][0-9])?|nge[0-9]|lo:[0-9]+',
# the name for the ethernet interface. eth0, eth1:1, lo:0 nge0 for Solaris.
```

to

```
'enet_if' => '1|eth[0-9](?:[:.][0-9])?|nge[0-9]|lo:[0-9]+|eri[0-9]+',
# the name for the ethernet interface. eth0, eth1:1, lo:0 nge0 for Solaris.
```

3.5 Special circumstances and configurations

There are some configurations that require a little care when defining them in DACS.
For a discussion of master and standy services see Configuring MASTER/STANDBY services section 7.9. For a discussion on configuring VMware hosts or Solaris Zones see Configuring Solaris Zones or VMware guest OS section 7.8.

3.5.1 Deploying multiple redundant servers

If you have multiple hosts that duplicate a service for redundancy, e.g. multiple primary NTP servers, or primary and backup mail delivery hosts you may need unique configurations for those hosts. To do this don't define an 'NTP' service, instead define numbered services: NTP1, NTP2, NTP3. Then the hosts associated with these services can be extracted using dbreport for generating configuration files.
This is particularly useful when you have multiple servers with different configurations and the configurations can run on any server. This also works well if it doesn't matter what client uses what server, or if the client will use all the servers.
If you have a single server at a site, and clients must use that site's server, using numbered service definitions can still be useful. But you may be better off defining a single service (E.G. NTP) and using class operations in rdist to select the service that is located at a particular site. So instead of NTP1 you would use the class of NTP_HOSTS intersected with site_lax1_C_HOSTS to identify the site specific ntp server. Now if there were two site specific NTP servers you could use:

- NTP1_HOSTS intersected with site_lax1_C_HOSTS and
- NTP2_HOSTS intersected with site_lax1_C_HOSTS

to distinguish between the two unique NTP servers at a site.

3.6 Standard database reports: wiring, asset, labels

DACS comes with a set of standard reports that you can access from the Config/database directory. The inline documention is shown below. See the online html documentation in Config/docs/database/Makefile.html for other information.

3.6.1 Obtain a report of all your assets in the database

```
assets - series of targets for generating asset reports
SYNOPSIS
  cd Config/database; make assets.csv assets.html assets.print
PURPOSE
This target generates asset listings in different
formats. For print, it generates latex that is converted
into postscript output suitable for sending to a
postscript printer. HTML output is in a tabular
format. CSV format can be imported into excel or other
spreadsheet program for analysis.
```

It lists the following fields in a table:

- Tag (inventory) number
- Name (sort field)
- Description
- Location

You can add more fields by editing the `report/report_assets` shell script.

3.6.2 Print labels to apply to hosts

```
NAME
   label - a target for generating host labels.
SYNOPSIS
  cd Config/database; make labels.print
PURPOSE
This target generates labels for placing on equipment.
It generates latex that is converted into postscript
output suitable for sending to a postscript printer.
```

3.6.3 Obtain a report of the wiring layout for your systems

```
NAME
   wiring - series of targets for generating wiring reports
SYNOPSIS
  cd Config/database; make wiring.csv wiring.html wiring.print
PURPOSE
This target generates a wiring report in different
formats. For print, it generates latex that is converted
into postscript output suitable for sending to a
postscript printer. HTML output is in a tabular
format. CSV format can be imported into excel or other
spreadsheet program for analysis.
```

It produces three tables listing:

- machine name
- contact person
- location
- wall_plug
- hub
- hub_port

for every machine. One table is sorted by machine name, another by hub/hub_port and the last by wall_plug (which is also used for specifying the label on a cable in a wiring closet or rack).

Chapter 4

The Version Control System (VCS)

DACS should be able to work with both subversion and CVS, however the CVS support has not been tested. The original version of DACS (known as Config) used only CVS. DACS changed to subversion (svn) because it was difficult to change the directory structure of the CCM repository with CVS. As a result reorganizing the repository to improve workflow is easier using subversion. There are some issues with subversion if you plan to implementing separate work and production CCM trees. The existing workarounds will be discussed below. Using subversion also complicates vendor version handling if you wish to track the vendor's original files. For most sites however, these advanced operations will not be used and subversion will work just fine.

4.1 What does the VCS provide

DACS incorporates a VCS into it's workflow to allow:

- recovery from bad configuration changes. Since it records every known state of the configuration tree, it is possible to roll back to a prior working state.

  While it is possible in an emergency to distribute a configuration tree that is not stored in the VCS, a number of features in the Rdist script have to be explicitly bypassed.

- validation of files before check-in. This helps prevent incorrect files, which would break working systems, from being checked into the VCS.

- collection and review of documentation on configuration changes. In addition the documentation (change log notes) can be validated. For example you can require a trouble ticket identifier to be included in the log to link the change back to the requirements that made the change necessary. It can forward the log message from a check-in messages into the associated ticket(s) to reduce the documentation burden. Also it can notify other admins by forwarding the log and and difference report to a mailing list. This allows post check-in peer review of changes.

- delegation of access to files in the VCS. Specific users can do their jobs more efficiently by having change access to specific files.

Also as an option it provides support for:

- separate production and work trees to enforce a process for deploying changes. This prevents un-reviewed changes from being pushed to production.
- tracking changes to vendor supplied files. This makes upgrading to new OS releases or newer versions of software easier and less prone to error.

4.2 Which VCS

There are currently two supported version control systems:

- subversion
  - makes moving directories and other restructuring of the CCM tree easier as it also preserves history.
  - is currently tested and deployed in a working DACS installation
  - is more secure access model using ssh
  - allows easy replication of ssh repository for redundancy (using svnsync with svn 1.4 and newer)
  - is a newer VCS and is still under active development
  - makes a split test/production tree a little more difficult to implement. This is still an area where work is being done in DACS.
  - makes tracking vendor copies of files different (some say more difficult) than in CVS.
- CVS (concurrent versions system)
  - makes implementing a split work/production tree is easier and allows easy access to a full history of changes.
  - provides better support for tracking releases of vendor files
  - is a mature VCS that has been heavily used
  - makes it difficult to move directories and preserve history.
  - is currently untested in DACS
  - has a less secure access model
  - has no direct support for maintaining redundant repositories

You may want to read Tracking vendor releases section 4.5.3 and Workflow in a split work/production implementation section 4.5.1 located below before selecting a VCS. If you don't plan on using split work/production or using vendor branches svn should be your preferred choice.

4.3 VCS anatomy

When discussing a VCS there are some terms you should know.

repository the directory where the VCS stores it's files. These files have a special format that allows the VCS to locate prior revisions of a file and track changes to the file structure.

working copy when you modify a file, you create a copy of the files in the repository as they exist at some point in time. Usually this time is the present. This copy of the files is called a working copy and is where you will perform your changes and testing before checking the changes into the repository. The CCM working copy is subdirectory of the repository tree. The working copy is a copy of the /Config/work tree hierarchy in the repository.

revision the recorded copy of a file or files at a given time

check-in the act of telling the VCS to store a copy of changed file(s) in your working copy as a new revision of the file(s) in the repository.

hook scripts these are scripts that are run by the VCS in response to user actions such as file check-in. They can validate changes to files (e.g. verify syntax), make sure that the log message has the proper information in it (e.g. a reference to a ticket) and other tasks.

These terms will be used throughout the rest of the documentation.

4.4 VCS setup

Because a CVS repository has not been tested in DACS, this section deals with the configuration and setup for subversion. But the issues discussed apply to both VCS's so you should read it and adapt the svn configurations to CVS (and send them to me) if you are going to use CVS.
This is not a primer on how to run svn or CVS. Many good primers and extensive documentation on your chosen VCS can be found on the Internet. See "Importing the DACS repository tree" section 9.1 for more details and examples to get started with a test DACS repository.
I suggest using subversion over ssh access with public key authentication. Also use a dedicated user to access the svn repository. This makes delegating access to files much easier as well as providing secure encrypted access to the repository. Setting up svn over ssh access is less complicated than other access mechanisms that provide the same security and delegation ability. The hook scripts that are provided in the DACS distribution assume you are using a subversion tunneled over ssh access method to a user account dedicated to subversion maintenance.
I strongly suggest making a separate DACS/operations subversion repository. Since sensitive items can be stored in the repository, ssh keys, passwords, community strings etc. maintaining a separate restricted access repository dedicated to DACS or system administration use is prudent.

4.4.1 Repository file structure

This is the repository setup that is supplied in the distribution.
The Config/work and Config/production must be self contained and (except for programs) not rely on any files from outside the tree. These trees can be checked out at any location and permit:

- rollback of changes to a prior working state
- testing changes in a working copy before check-in
- multiple independent copies of the DACS system for redundancy and load balancing

```
svn_root/Config/work -+- Distfile
                     +- Rdist (link to Config/bin/Rdist)
                     +- Config +- bin - Rdist, dbreport, filepp
                               +- database (replaces Database in Config)
                               +- distfile (replaces Dist in Config)
                               +- lib/config/* - support files

svn_root/Config/production - same structure as work. This is actually
                             a branch of the work tree that is used to
                             push tested and reviewed files to
                             production.

svn_root/Config/vendor/vendor_dir - each vendor_dir has a similar
                          format to the work tree. This tree is used to
                          enter new vendor copies of files into config
                          with a history. E.G. we need to change
                          =/etc/rc.d/init.d/sendmail=. We define a path
                          in work to hold this file
                          (=work/rc/sendmail=). Then we define the same
                          path under the vendor branch for that release
                          (=vendor/centos_4.2/rc/sendmail=) for example
                          and check-in the unmodified vendor file. Then
                          when a 4.3 release of the file is installed,
                          we check the 4.3 version into
                          =vendor/centos_4.3/rc/sendmail= and then
                          merger the changes between 4.2 and 4.3 into
                          the work copy. (See subversion docs for
                          more information on vendor branches.)

svn_root/SVN  - a directory containing hook and conf files for the
                DACS repository. Once the hook scripts are deployed,
                they will be automatically updated upon subsequent
                check-ins to this directory.
```

Figure 4.1: Tree diagram of the subversion repository structure.

Under the Config/work and Config/production trees you will set up directories that group files by task or relationship. The top level directory structure is replicated through to the distribution system. For example, the `ntp` directory can include all the files needed to deploy NTP across the enterprise. Because the `ntp` directory exists, there should be an `ntp` label in the distribution system that is used to allow selective update/distribution of just this directory when pushing files. The structure within the `ntp` directory is totally under your control. There may be subdirectories on the remote site receiving the files: site/lax1, site/mia1, or it can be a flat directory structure.

As another example, you may have an `ssh` directory that includes all the ssh configuration files and an `etc` directory that includes random single files that live under /etc. The files in these directories would be pushed by passing the `ssh` and `etc` labels to the distribution command Rdist.

Because of this coupling between the directory names and the labels in Rdist that select subgroups of files to push, the repository layout affects how your work is done. As time progresses

you often want to move files and directories around to put files that get changed as a group under the same directory hierarchy to make editing and distribution easier and faster. This is the primary reason subversion was chosen as the main VCS for DACS.
In the DACS distribution the following directories are supplied:

config_bin scripts used by the DACS system

cron sample cron scripts for various purposes

etc build system for individual host files, other assorted files

firewalls system to build an iptables based firewall system

ntp system to build ntp configurations for all hosts

pkgs system for software installation in a yum environment

services system for building a services auditing and enabling under Linux

ssh system for recording ssh keys and generating known_hosts file for distribution

users files for managing files associated with users

Some of these directories are used in the DacsExamples chapter and are referenced in examples throughout this documentation. Additional documentation on these directories and files are in the DACS repository distribution.

4.4.2 Sample SVN configuration and hook files

In the distribution there is a directory called SVN that includes the hook scripts and sample config files for an SVN based DACS release. These hooks scripts implement repository replication (if using subversion 1.4 or newer) as well as serving as an example of file and commit log verification. The hooks scripts include:

- a pre-commit script that
  - allows access if the check is a replication request
  - checks to see if the repository is locked from commits
  - checks to verify a non-blank commit comment that includes a reference to an RT ticket number (specified as RT:3452)
  - checks to verify the syntax of the users/sudoers/sudoers config file
  - checks to verify the syntax of executable scripts
- post-commit script to perform actions on the newly checked in files that:
  - updates the subversion control files for the repository (which are maintained under svn) and installs a new authorized_keys file for the repository owner/account from the subversion update
  - saves a commit log entry in the commit-log

  - exits the script if this check-in was due to a replication request
  - sends email about the commit to users
  - synchronizes the repository to its replicas if it is run on a master repository

- pre-revprop files that controls values of meta information (e.g. author, date) assigned to a check-in that
  - allows the change if it was called as a result of replication
  - allows addition of new values, but prevents changing of old values

- post-revprop file that handles completed value changes that
  - records property changes in the commit-log
  - synchronizes the repository to its replicas if it is run on a master repository

There is only one subversion configuration file that is used with the normal `svn+ssh://` access mechanism:

- `conf/authzfile` that defines default access rights compatible with DACS and includes authorization for the dacsuser repository owner.

and two configuration files specific to the hook scripts:

- slave_urls - contains the urls (svn+ssh://...) to one or more subversion repositories that should be kept in sync with the master repository

- authorized_keys - this is a copy of the authorized_keys file for the owner of the repository. The hook scripts will install this file in `~/.ssh/authorized_keys` so you can track changes to this file. The distributed copy of this file has invalid sanitized public keys, but it demonstrates the basic format and settings for the file.

To use these scripts, you replace the repository's `conf` and `hooks` directories with checked out copies of SVN/hooks and SVN/conf. See Importing the DACS repository tree/Subversion hook scripts section 9.1.1 for details on doing this.

4.4.3 SVN Access controls

For SVN versions greater then 1.3, the svn server (which is used when accessing the repository via ssh) can use access control lists. Before then only the svn Apache (HTTP protocol) module provided access controls. Because of this DACS supports only subversion releases newer than 1.3. If you must use a pre 1.3 release of subversion don't use delegation. There is no way to restrict read access to the files (using the ssh transport mechanism) in the repository. Pre-commit hook scripts can only prevent writing/updating of changes and don't restrict checkouts. Because of the sensitive nature of the data (e.g. passwords in config files, ssh keys etc) the ability to restrict read access is required. You could try setting up https access, but I consider that less secure tha using ssh access and given the nature of the files stored in DACS, I consider using https too much of a risk.

Access restrictions can be set up in any version of CVS using standard file/directory permissions in the CVS repository tree, but the restriction are not as easily controlled/audited as the subversion implementation.
The subversion access control file for a repository is located in the file `conf/authzfile` under the root of the svn repository.
If you are using the sample SVN files from the DACS distribution, the conf directory should be a working copy of the SVN/conf directory supplied in the DACS distribution.
The `authzfile` in the DACS distribution provides read/write capability to admins by default. A sample authzfile looks like:

```
[groups]
admins = you
others = somebodyelse

[/]
@others =
@admins = rw

[/Config]
root = r
@admins = rw

[/Config/work/mail/address]
others = rw
```

People in the admin group can read/write everything (including the SVN control files). Root has read only access to the entire /Config tree so it can retrieve updates using Rdist. Because root is not a real person it does not have write access. Users in the others group are delegated read and write access to files under the mail/address directory, see below for further info on delegation.
To add a new admin, you add the user name of the admin to the admins line changing:

```
admins = you
```

into

```
admins = you, me
```

For details on the format of this file I refer you to the subversion documentation at:
`http://svnbook.red-bean.com/`

4.4.4 Delegation of access to a non-admin user

A feature of DACS is the ability to delegate system changes to other people. For example a developer may need to change the hosts file on his system.
It would be efficient for him to change this file without having to involve the administrators. Yet if something goes wrong, the administrators want to be able to see what was changed and have an easy way to roll back the change.
Delegating access to this file to the user, produces exactly this effect. The user can change (and deploy using `sudo /config/Rdist`) the hosts file for his system. The VCS underlying DACS provides the ability to see what changed and roll the change back. The administrators control the

file generation and distribution mechanisms/rules so they have the ultimate authority on what changes the user can perform.
To allow non-admin users to use DACS, you have to allow the user access to a portion of the DACS tree. SVN access controls are directory based, you can't set access controls on a single file (well you can, but it makes things a bit tricky). Instead create a portion of the tree for the user to work in. E.G. create the directory `etc/hosts/fredchanges`, where `etc/` is at the top level of the DACS work tree.
Then add:

```
# allow fred access to modify the hosts file
# note that this is a directory with a single file (hosts) in it
# that is validated by the build system and merged into hosts files.
[/Config/work/etc/hosts/fredchanges]
fred = rw
```

(Note that /Config/work is the location of the DACS tree in subversion and the working copy that is checked out is the tree starting at /Config/work.)
Fred will be restricted to reading and writing files under the `fredchanges` directory. Note do not allow fred to change files in `<svn_root>/Config/work/etc` because the presence of a `Makefile` in that directory will execute the Makefile commands as root (see DacsBuild for details on the build/make mechanism). If you delegate access to any top level directory under /Config/work you have given the delegated user the ability to run any program as root on your DACS server. As long as the users have access to a directory two levels down from the root, you control what will run by properly constructing the Makefile that triggers the build system.
If the user changes a file that needs to be processed, a DACS administrator will have to create the Makefile to generate files on behalf of the delegated users. This is a good way to delegate access to a portion of a configuration file. The user edits a file and the contents of that file are processed into a larger configuration file.
The administrator is responsible for vetting any files delegated to a user before processing them. Using `su` or `sudo -u` you can generate files as the delegated user if you wish. However the safest course is to have the file generation mechanisms controlled by the admin and only allow the user to generate input data. Note that the same warnings apply to any check or other validation scripts that run as part of the VCS hooks.
Once the delegated file is in place, it can be used just like any other file. It should have entries placed in distfile.base that push the delegated file when it changes.

4.4.5 Setting owners, groups and permissions

Neither CVS nor subversion preserve owner/group or mode information. Both require external support to modify the CCM tree to set these. The build system performs the operations required to do this.
Under CVS, the commands run by the build system are in a shell script that is managed like any other file under CVS. It is invoked by the Makefile in the build system and sets the permissions.
Subversion allows you to set properties on files which stay with the file if it is moved. Using svn properties, you can store the information with the file and use the build system to extract and apply this information. Since the properties move with the file, you don't need to update a script to keep the permissions after renaming a file. The DACS build system uses the svn properties:

- svn:owner
- svn:group
- svn:unix-mode

that was proposed on one of the subversion mailing lists to store the properties. The value of these properties is passed to chown, chgrp or chmod respectively. So using
`svn propset svn:owner bin thisfile` and then checking in the change will allow the command in the build system to execute a 'chown bin thisfile'. Since the arguments are passed to the underlying tools, you can also use numeric values (uid/gid) in case the user doesn't exist in the password file on the DACS master host.
See Setting modes, owner and groups section 5.4 in the DACS build chapter for the details on creating a Makefile that sets file access from subversion properties.

4.5 Workflow under the two VCS

With either VCS's a typical update cycle consists of:

- perform a checkout of the portion (or whole) CCM work tree.
- update the files using the editor
- verify your update
- check the files into the VCS with a log message
- run Rdist to pull the changes from the work tree and distribute

This is the simplest mode of operation where there is only one tree that operates as both the work and production tree and is pushed to all hosts. This works well for development sites or other sites where the overhead of a review isn't needed.
It also work well when you have a group of senior admins who rarely if ever screw up and can quickly minimize the effect of the failure by reverting the change. Also this works well at educational sites as messing up 100+ workstations is a really good lesson, and the consequences can be reversed quickly by rolling back the change and redistributing.
However for some sites this is insufficient and they want to implement a more formal process. One such process uses two different checked out trees:

work this tree is where all the changes occur. When it is checked out to make a master tree, the Rdist command recognizes the location and will restrict the hosts that can be updated from the tree. This is also called the test tree as it is used for testing changes to the configuration.

production this tree gets copies of files taken from the work tree that have passed some validation mechanism/process to be promoted to production status. The Rdist command recognizes this tree and permits it to update all the hosts that are defined.

4.5.1 Workflow in a split work/production implementation

This implementation defines two separate DACS master trees. The work tree and the production tree. The work tree is used to make operational updates to a test network or set of test hosts. The production tree makes updates to systems that are in production. Files must be promoted from the work tree to the production tree.
All changes are committed to the work tree, so you start by having a working copy of the work tree checked out, making a change to the work tree and checking the change in. In this respect it is identical to the simple operation mode.
When you run Rdist, it will not be from /config/Rdist, but instead from an alternate master tree: /config.test/Rdist. The /config.test tree is set up (via settings in Rdist) to only allow updates to hosts that have the TESTHOST service defined in the database.
/config.test/Rdist will:

- update the test distribution tree
- generate new copies of any files needed by the target hosts
- push the changes to the test hosts

When you are satisfied that your changes are correct. The changes are reviewed and promoted into the production tree. Then the production master tree (under /config) is used to Rdist the changes to all servers. If the promotion isn't done, no change occurs to the production tree.
The split of workflow across work (or test) and production distribution trees is useful since it allows for the enforcement (via svn hooks or external script) of particular conditions before the files are pushed to production. For example:

- the person who checked the changes into the work/test tree can not promote the changes to the production tree
- only a subset of people can promote files to the production tree

This allows the person doing the promotion to verify the changes before they are placed in production.

4.5.2 Promoting from work to production

I will describe the split work/production setup under CVS first as it is more straight forward. Then I will describe this operation using subversion for the repository.
CVS has what is known as floating tags. These tags mark a point in time for a file under the CVS system. They can be assigned to different versions of different files to bring them all together as a single named entity. So revision 1.6 of one file can be tagged the same as revision 1.49 of another file. By moving the "PRODUCTION" tag to specific revisions of files you mark the promoted files. To implement this you need:

- a CCM working tree (/config) that is stuck (a "sticky tag") to the PRODUCTION tag (`cvs co -r PRODUCTION configtree`)

then you execute the following command sequence:

- tag the files in the work tree that are to go into production with two tags (using cvs rtag): `PRODUCTION` and `PRODUCTION_uniqueid`
  where `uniqueid` may be a datestamp or some other unique identifier.
- note that this tagging occurs in the repository with no working copy. So there is no way of getting a file that wasn't checked into the work tree. Hence you should always be able to revert production to a known prior tested state.
- then a `cvs update` (done by Rdist) of the production tree will install the newly tagged PRODUCTION files and distribute them.

using this method you have:

- a history of all the files/changes moved into production identified by the PRODUCTION_uniqueid tag on the files.
- the ability to see the entire history of the current files in production by using cvs log on the file in the checked out production tree.
- the ability to rollback a change by finding the prior PRODUCTION_uniqueid tag and re-tagging that revision with the PRODUCTION tag.

This works the way it does because tags in CVS exist only on the revision axis of the repository. They mark points in time of a single file.
There are two different ways to do this under subversion. The preferred way is discussed first.

4.5.2.1 Promoting from work to production (svn method 1)

In contrast to CVS tags, subversion (svn) tags are not based in time, they are based on copying the files at a point in time to a new name. So promoting a file to production in svn requires:

- a CCM tree checked out that is stuck to the most recent (HEAD) of the PRODUCTION copy of the files (`svn co svn+ssh://dacsuser@dacsmaster/repo/Config/production` rather than the work tree located at:
  `svn co svn+ssh://dacsuser@dacsmaster/repo/Config/work`)

then you have promote a file by:

- deleting the current copy of the file in the production tree and coping in a replacement file from the development tree that you want to promote to production. Note that all these operations are strictly in the repository
  - `svn rm svn+ssh://dacsuser@dacsmaster/repo/Config/production/example/file`
  - ```
    svn cp \
    svn+ssh://dacsuser@dacsmaster/repo/Config/work/example/file@version \
    svn+ssh://dacsuser@dacsmaster/repo/Config/production/example/file
    ```
  - this creates two separate revisions in the repository (one remove and one copy) unless you use a tool such as mucc to make it into a single atomic revision/transaction.
    ```

– since these operations are all in the repository, there is no way to promote a file that wasn't recorded in the repository, which is a good thing.

Performing an `svn log` on the file under /config (i.e. the production tree) displays the history of the file and because of the copy operation it also includes all the log information from the file when it existed on the development tree. So you can see the history of the file in production. So far so good, but what happens when we want to find out what the prior production release was and roll back to it?
Now we have an issue. We had to delete the prior copy in the production tree otherwise the copy would fail. We can view the history of the current file in the production tree, but we lose the ability to tell what the prior revisions were that went into production. We can retrieve this info by specify peg revisions on the fils in the production tree, but it is not straight forward since prior production releases have been deleted from the history accessible from the head of the production tree. Hence there is currently no simple way of seeing all the prior revisions of the production tree. In theory we can rollback just the last few commits to restore the production tree to the earlier state.
However we have done multiple remove/copy cycles for all the files being promoted (unless mucc was used) and each one is a new revision in the tree. Plus there may be other promotions that have occurred that we don't want to delete, so undoing just a few promotions is more complicated that it may seem.
So this method isn't as clean as one would hope. There is work in progress on promoting files using this method with mucc to provide a single commit promotion of multiple files. There is also a prototype of a tool that can scan backwards in the production history to ease recovery. But the tools aren't ready yet.

4.5.2.1.1 The manual subversion steps to promote a file If the production config tree (at `/config`) is setup to sync to the production branch, you need to get the copy in the work branch moved (promoted) to the production branch. Before doing this identify the version of the file(s) that you want to promote using `svn list`. In this example we will promote a new Rdist command:

```
svn list -v svn+ssh://dacsuser@dacsmaster:/repo/Config/work/users/sudoers/sudoers
         628 rouilj               28299 Jun 01 18:35 sudoers
```

This is the `sudoers` file that exists at version 628 of the repository. Remove the old sudoers file in the production tree and copy in the new file with:

```
svn rm svn+ssh://dacsuser@dacsmaster:/repo/Config/production/users/sudoers/sudoers

svn copy svn+ssh://dacsuser@dacsmaster:/repo/Config/work/users/sudoers/sudoers@628 \
 svn+ssh://dacsuser@dacsmaster:/repo/Config/production/users/sudoers/sudoers
```

Once this is done using `/config/Rdist` will pull the new sudoers file and distribute it. The person doing the copy is expected to verify that the new Rdist file has valid approved changes and is suitable for production use.
Some sites prefer the person who committed the file to be promoted (rouilj in this case) shouldn't be able to promote it to production. This should be enforceable using pre-commit hooks, but I have not yet attempted to do this using subversion, but some combination of:

- `svnlook changed --copy info` to find the source path and version of the files in the work tree
- `svnlook history source_file` to find the revision the source file was changed in prior to the copy operation
- `svnlook author revision` to find the author of the last change

should allow the hook script to verify this info. However this may be more easily done using an external script.
In this scenario you may also want to set up separate permissions for the production and work trees as in this example:

```
[groups]
admins = you, somebody
promoters = boss, you

[/]
@others =
@admins = rw

[/Config]
root = r
@admins =

[/Config/work]
@admins = rw

[/Config/production]
@promoters = rw
```

You can check into the work tree and promote to the production tree, but somebody can only change the work tree and boss can only promote things to production.

4.5.2.2 Promoting from work to production (svn method 2)

The second way of moving changes from the work tree to the production tree preserves the production history allowing a fast rollback, but makes accessing the log information associated with the work tree more difficult. Plus it introduces the possibility that a production release may not consist totally of files from the work tree in the repository. In addition it requires a working copy of the tree to be checked out. Because it is not a recommended mechanism, it's not discussed in as much detail as the first method.
When using this method, we still have our CCM master tree (/config) that is following the production tree. We do the following:

- check out a local working copy of the production tree
- perform an `svn merge` from the revisions on the work tree to get the changes between the version in the production tree and the version you want to promote to production.

- check-in the new file version to the production tree.

Now `svn log` provides you with a list of versions on the production tree, so you can perform a reverse merge and get back to a working configuration, but you don't have access to the change log info that was recorded in the work tree without manually tracking revisions. Because this mechanism requires a working copy, it is possible to check-in a file that never existed in the development repository tree (and was therefore never tested). However all the promotions are committed as a single revision which makes rolling back a promotion easier.

4.5.3 Tracking vendor releases

Similar to the issues with a split work and production CCM tree, tracking vendor releases of files is also a little trickier in svn than CVS. This method is used to track changes to the vendor supplied files that are DACS managed. By tracking the changes, you can generate differences between the prior and current vendor files and apply these differences to your working trees to get updated configuration file settings, options etc.
As we did when discussing a split work/production tree, an example using CVS will be discussed first followed by a subversion example.
In CVS a branch is not a separate file/location in the repository. It is an alternate time sequence for a file. So when you move the file in CVS, you also move the vendor branches. (footnote: CVS's standard rename/restructure mechanism causes problems when checking out prior revisions of a tree as the original and new locations for a file would often be checked out so CVS was not without issues in this regard either). Consider this scenario under CVS:
You create a copy of the original file in the centos_4 vendor branch. This stores the virgin copy of the file as shipped by the OS vendor at release 4.3. You also tag this file so you can find it later. Then you check out this file and use it as the base file in your work tree. You modify the file to work at your site and push it to all the hosts. Then you do a little restructuring moving the file from the etc directory to the dns directory.
Now you have a new OS release. Let's assume you started at OS version 4.3 and now move to 4.5. This file in the distribution has changed between releases 4.3 and release 4.5.
So you check out the centos_4 vendor branch and copy all the files from the 4.5 OS release to their corresponding location in the centos_4 vendor tree. Then you check-in the new 4.5 populated vendor tree and tag the changes as the centos_4.5 release. You can now use `cvs diff` between the 4.3 and 4.5 vendor branches to review the differences between them. Also you can merge these differences into your working copies. So far so good.
In subversion you would do much the same except a subversion branch is a separate file path. So moving the working copy of the file doesn't move the files within the branch, that must be tracked manually for each vendor branch. Initially:

- the vendor branch/tag would live in vendor/centos_4.3/etc/file
- the distribution copy would live in work/etc/file

then you move the distribution copy to work/dns/file, where is the vendor copy? You guessed it, still at etc/file. Sadly it's very easy to forget where things are stashed, and it's made more difficult because your dns/file may have many ancestors:

- changes from the vendor/centos_4/etc/file
- changes from the vendor/centos_5/etc/file
- changes from the vendor/solaris_10/etc/file

etc. depending on how universal the file is (e.g. think of the /etc/resolv.conf template file that incorporates changes from a number of systems). In CVS all the vendor tags/vendor branches move along with the base file so it makes locating the branches easier.
In subversion you create a copy of the original file in the centos_4.3 vendor tag which stores the virgin copy of the file as shipped by the OS vendor at release 4.3. Then you copy this virgin file (using the svn copy command) into the work tree. This copy operation records the origin of the copied file telling you where it came from. So you can get a mapping between the work and vendor trees by viewing the history of the first revision of the work file. You modify the work file as needed for your site and push it to all the hosts. Then you do a little restructuring moving the file from the etc directory to the dns directory.
Now you have a new OS release. Let's assume you started at OS version 4.3 and now move to 4.5. This file in the distribution has changed between releases 4.3 and release 4.5.
So you check out the 4.3 vendor tree and copy all the files from the 4.5 OS release to their corresponding location in the 4.3 vendor tree Then you check-in the new 4.5 populated vendor tree as the centos_4.5 tag preserving the original copies of all the files as they exist in the 4.5 OS release. You can now use `svn diff` to generate a list of differences and merge these differences into your working copies. So far so good. Except you see a diff for etc/file, where does that file reside now? You have to search the work tree looking at the first revision of each file (or have recorded this information) to find the mapping from the vendor branch to the working branch. Note there is no mapping from the vendor branch to the work branch, the mapping is from the work file to the vendor file.
Regardless of whether you use CVS or subversion for this there are some issues to overcome. Most sites I know of don't maintain separate vendor branches at all and just walk the work trees and manually try to find differences in versions and incorporate the changes.
See DacsExamples for a discussion of creating and using a vendor branch for the `/etc/services` file. The subversion manual
`http://svnbook.red-bean.com/en/1.4/svn-book.html#svn.advanced.vendorbr` provides more detailed information on handling vendor branches under subversion.

Chapter 5

The Build System

The build system uses gnu make to generate files. A replacement can be substituted, but only gnu make has been tested. The distribution system uses the build system to generate the control file that determine what hosts receive the distributed files, so you may just want to use gnu make for everything.

5.1 What does the Build System provide

For some sites the build system may not be needed. However a lot of configuration files have information (hostnames, ip addresses) embedded in them. If you don't use the build system this information is usually specified at least twice:

- Once in the DACS database where it drives the distribution system
- In one or more configuration files

For example the files `/etc/resolv.conf` and `/etc/dhcpd.conf` use information about the DNS servers in your installation. When you change the DNS servers in the DACS database, you should either:

- generate new copies of these files with the new DNS server assignments or
- verify that these files still have valid DNS information

It is very easy to miss changes to these files and have problems weeks or months later when a host using DHCP reboots and it no longer has a valid dns server. Even worse is when only one of the three dns servers is correct in resolv.conf and the one functioning DNS server reboots or crashes leaving most of your network without any DNS service. Using the build system you can implement either option and eliminate the problem entirely.
So the build system:

- generates configuration files for hosts using information from the database or information from other configuration files contained in the DACS CCM tree to generate correct and consistent operation.

- validates configuration files against other files in the DACS CCM tree to verify that the information is correct.
- provides a mechanism for generating individual configuration files for every host based on the attributes of that host. E.G. iptables files can be generated that allow access to port 80 only if the host is running the Apache web server.
- provides a mechanism to generate output files from input files that have change rights delegated to non-admin users. This allows non-admin users to modify portions of a larger configuration file.
- sets owner, group and mode of files in the DACS CCM tree so that they are installed on remote machines with proper settings.

5.2 Build system setup

To trigger the build system place a file called `Makefile` under any of the top level directories in the DACS CCM tree. Rdist will automatically call `make all` or `make verify` (when run in distribution or verify mode respectively) using the Makefile if the directory name matches the selected labels that are being distributed (see "Distfile Label Types and Special Purpose Labels" section 6.4.2 for details on the link between directory names and labels).
The Makefile rules can generate all the required files itself, or it can invoke other Makefiles using `$(MAKE) -C <subdir>` to generate files in subdirectories. This submake form is useful if you have multiple independent subdirectories under your top level directory.

5.3 Makefile template

The Makefile should have a verify target in it as it will generate files for pushing during a verify phase. A prototype gnu make Makefile is:

```
all: [default_files_to_generate]
verify: all

.PHONY: all verify
```

Replace `[default_files_to_generate]` with any make(1) targets that you want to build. See the documentation for make(1) for further details. Also in the DACS release there are Makefiles that you can use as examples.

5.4 Setting modes, owner and groups

Permissions, ownership and group membership are not preserved by CVS or subversion. To handle CVS permission setting, create a shell script, or a target in the makefile. E.G.

```
all: security

security:
```

```
chmod 755 file1
chgrp 123 file1
chown 456 file1
chown bin file2
        ...

.PHONY: security
```

(Note that all the command lines begin with a tab).
To perform the same thing with subversion we use the subversion properties:

- svn:owner
- svn:group
- svn:unix-mode

on each file. See section 4.4.5 for more details on this The properties are set using the `svn propset` command like so:

```
svn propset svn:unix-mode 600 <file/directory>
```

The file properties are updated on an svn update and extracted using the recursive propget (property get).
To do this, create a Makefile in the first subdirectory under the root of the tree. E.G. "firewalls/Makefile" or "Config/Makefile". These makefiles will trigger the makefile execution functionality built into DACS.
Then modify the makefile so that it includes the following:

```
# define location of root of the config tree.
RELATIVE_TOPDIR=..

# include this makefile to define:
# security target that will set owner/group/perms from svn properties.
# also individual modes, user and group targets
include $(RELATIVE_TOPDIR)/Config/lib/config/Makefile.permissions.include

all: security

verify: all
```

This uses the provided include file that has all the command to perform the maintenance of permissions, user and groups. By running the `security` target from the `all` or `verify` target you set permissions, ownership and modes.
If you are in your own working copy of the CCM tree, you aren't operating as root and therefore make will fail when trying to set the owner or group. If you set the environment variable: `NOCHOWNCHGRP` to any value, then `make security` will skip the group and owner modification steps only if you are not running as root. However it is still a good idea to unset the variable before running any Rdist command from a master tree.
In addition to the security target, you can use:

- make .set_modes - to set just the modes of the files
- make .set_group - to set just the groups of the files
- make .set_owner - to set just the owners of the files

Note the target names start with a dot. In general they aren't used explicitly and the leading dot is used to indicate that they are internal targets used by `security`.

5.4.1 Implementation notes

Arguably this mechanism should be placed into the Rdist wrapper or be maintained by a Makefile at the root of the DACS working tree so that each top level directory doesn't need to deploy a Makefile just to keep permissions up to date. However the current implementation permits checking out a portion of a tree and being able to test it.
This mechanism wastes resources and time because there is no easy way to find out if the permissions on the files have changed. The only way to find out is to analyzing the output of the svn update command. Analyzing the output takes longer than just setting the permissions every time. Since a change in permissions doesn't update the time stamp of the checked out file other make(1) based attempts to work around the problem are foiled.

5.5 Reducing dbreport invocations (using a per host cache)

When you generate files that are different for each host, the information in the DACS database is often used to select the contents of the generated files. By convention, the generated files are placed in the `dist` subdirectory and named for the host. So the copy of `/etc/hosts` generated for the host `a.example.com` would be generated to: `etc/hosts/dist/a.example.com`. When using gnu make you would use a pattern rule like:

```
dist/%: ../Config/database/db Makefile
some commands
```

that says that any file in the `dist` subdirectory can be generated by this rule. (In the target `dist/%`, the under the `dist` subdirectory, so it would be invoked for `dist/a.example.com`.) Also this rule states that the files in `dist/` need to be rebuilt if the database `../Config/database/db` or the `Makefile` change. The down side of this is any change to the database will result in *all* the dist/* files being regenerated. This happens even if the database change doesn't change the configuration for any host (e.g. a comment was added to the database file).
Also the database report on each individual host may be needed by multiple Makefiles. Without some shared caching mechanism, each Makefile will have to run the commands needed to generate the per host database report. Since a lot of time is spent generating per host data from the database, a shared caching mechanism was added to the build system that generates the per host information once per database change and detects changes to that information before using it.
To use the caching mechanism add the following at the top of the Makefile (this example is taken from the Makefile that generated iptables firewall configurations for each host):

```
# define location of root of the config tree.
RELATIVE_TOPDIR=..

# define this makefile for dependencies below.
# this can be modified by included makefiles to
# be a list of all makefiles
ALL_MAKEFILES:=$(word $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST))

# override the default selector given to dbreport.
# VALID_HOSTS for this makefile are hosts that do not have the NOFIREWALL
# uses clause and do not have rdist set to no.
DBREPORT_SELECTOR:=-n 'rdist:/no/|uses:/\bNOFIREWALL\b/'

# include this makefile to define:
# VALID_HOSTS, default value of SPEC_HOSTS and rules needed to create
# .sum files and cached info files.
include $(RELATIVE_TOPDIR)/Config/lib/config/Makefile.md5cache.include

# Create the list of files to be created. Select elements of the
# (optionally) defined SPEC_HOSTS macro which defaults to all VALID_HOSTS.
per_host_files:=$(addprefix dist/,$(filter $(SPEC_HOSTS), $(VALID_HOSTS)))

# Define the default targets
all: $(per_host_files)

verify: all

dist/%: $(ALL_MAKEFILES) .sum/% <other files, templates ...>
build commands using the per host database cache files
located at $(HOST_CACHE)/$* for each host
```

This implements an md5 checksum mediated cache by setting three variables:

RELATIVE_TOPDIR the location of the root of the DACS CCM tree relative to the Makefile

ALL_MAKEFILES this setting should be copied verbatim and includes the paths to all the makefiles involved in generating the files.

DBREPORT_SELECTOR the setting of DBREPORT_SELECTOR is optional and by default includes all the hosts in the DACS database that do not have their `rdist` value set to `no`). It is passed to the dbreport command (see DacsDatabase) to refine the list of hosts that are generated. In the example above it removes hosts not under DACS control and that do have the `uses` ... NOFIREWALL ...= value set indicating that they do not use a unique firewall from DACS.

and including the `Makefile.md5cache.include` makefile from the library.
The only change to the original `dist/%` rule is to make the target depend on `.sum/%` and `$(ALL_MAKEFILES)` rather than the DACS database and the local Makefile. So when creating the file `dist/a.example.com`, make will verify that the file `.sum/a.example.com` is up to date as well as the list of makefiles.

The firewalls directory included in the distribution is one example of using this caching mechanism. The services directory provides a second example. Also there is a Makefile.cache included in the ntp diectory that doesn't use per host files, but still can use the mediated cache to reduce the number of database reports that need to be generated.

5.5.1 VCS Interaction

Since the `.sum` and `dist` directories are output directories for files generated by the Makefile, their contents should always be reproducible by checking out a prior release of the tree and re-running the build system. So the contents of the `.sum` and `dist` would NOT be checked into the VCS system, but the directories themselves are in the VCS.
The VCS system needs to be told to ignore any files located under the `.sum` or `dist` directories otherwise one of the safety interlocks in DACS will stop file distribution. This safety interlock checks to make sure that there are no modified or unrecognized files (i.e. the files under `.sum` and `.dist`) in the checked out tree. By setting the svn:ignore property for the `.sum` and `dist` directories to '*' any files in the .sum or dist directory are made invisible to the VCS system. Since these files are recoverable from a prior version of the VCS tree it is safe to tell the VCS to ignore these build artifacts.
To set these directories up, you should make the `.sum` and `dist` directories and add them to subversion (`svn add .sum dist`). Then tell svn to ignore the contents of the directories using `svn propset svn:ignore '*' .sum dist`.
You can do the same with CVS, but rather than using `svn propset`, you add a `.cvsignore` file under the `dist` and `.sum` directories with the contents of '*' to ignore all the files.
Then you can check-in the directories under either subversion or CVS to finalize the change.

5.5.2 Adding new data to the host caches

To add new data to the host cache modify the script
`Config/lib/config/make_per_host_database_cachefile`.
For example, suppose you have an ftp service running on a host at a unique IP address (192.0.2.72), and you need to modify the firewall rules to allow access to the ftp server only if an ftp request come in for the 192.0.2.72 address. So the problem becomes what is the address of ftp.example.com assigned to the server s1.lax1.example.com?
To solve this, get a list of IP's where the base host is the current host (s1.lax1.example.com) and the child host includes the FTPEXT value.
`dbreport -f ip -s 'base:/s1.lax1.example.com/|services:/\bFTPEXT\b/'` (See section 3.4 for info on using dbreport).
Add this command to Config/lib/config/make_per_host_database_cachefile and append it to the cache file defining the macro FTPEXTSVCIP.
This looks like:

```
# if the host provides FTP service, capture that external IP address
# and record it.
FTPEXTSVCIP="`${DBREPORT} -f ip -s "base:/^${HOST}\$/|services:/\bFTPEXT\b/" \
     -n 'isbase:/no/' ${DB}`"
if [ -n "$FTPEXTSVCIP" ]; then
echo "#define FTPEXTSVCIP $FTPEXTSVCIP" >> ${CACHEFILE}.$$.tmp
fi
```

Then in the firewall rules file, if the FTPEXTSVCIP is defined include a rule like:

```
#ifdef FTPEXTSVCIP
  #if FTPEXTSERVICE ne ""
-A IN -d FTPEXTSVCIP -p tcp -m tcp --dport 21 -m state --state NEW -j ACCEPT
  #else
  #undef FTPEXTSVCIP
  #error HOSTNAME has a null value for FTPEXTSVCIP
  #endif
#endif
```

where we add a little error checking to make sure that the address is not null. This adds the rule:

```
-A IN -d 192.0.2.72 -p tcp -m tcp --dport 21 -m state --state NEW -j ACCEPT
```

to the iptables configuration file `dist/s1.lax1.example.com`.
Note that we originally set this up for s1.lax1.example.com, but it implements the general rule:

> If a system provides the FTPEXT service, it must have a unique IP address for that service, and only ftp requests to that IP address will be accepted.

If the FTPEXT service is moved to a different host in the database, then the firewall rule also moves with it and is removed from the old server. If an additional FTPEXT server is set up, it has to be set up according to the specification above or the FTPEXTSVCIP will be empty and the build will fail, preventing the distribution of incorrect firewall rules.

5.5.3 The internals

This section describes how the md5 mediated caching mechanism works and why it was devised along with performance improvement numbers. If you are interested you can read this section but it is likely to be of little interest unless you are quite familiar with make.
Normally, you use this dependency chain (called the direct dependency mode):

```
        dist/A.file -------+
                           +--- database
        dist/B.file -------+
```

This causes a build to start for both A.file and B.file every time the database changes regardless of what info changed. In Makefile syntax this is expressed with:
`dist/A.file dist/B.file: database`. Now we could use (the cache mode):

```
        dist/A.file --- A.report ---+
                                    +--- database
        dist/B.file --- B.report ---+
```

where the A.report contains the info needed from the database to create A.file. When the database updates, the A/B.report files can be generated to a temporary location, compared to the original report files and if they are different, the new copies of the report files replace the original copies.

So when the database entry for A is updated, the A.report file is replaced but B.report isn't. So A.file is regenerated while B.file isn't.
So this kind of does what we want, but not really. Here's why. The majority of the processing time is spent generating the A.report and B.report files. So if I update system A, the make runs, generating a new A.file and A.report. it leaves B.report alone since no data has been updated. Now we do another make without any other changes, (which is automatically done by the DACS system when you go to verify or update the corresponding label). The file B.report is older than the database, so we regenerate the report and find once again that B.report doesn't need to be updated. We keep doing the needless work of regenerating the unchanged B.report information to prevent performing the update of file B.
Other make systems (makepp (`http://makepp.sourceforge.net/`), cons (`http://www.dsmit.com/cons/`)) don't depend on timestamps to determine if B.file should be regenerated. They record the checksums of the inputs for B.file (e.g. B.report, Makefile ...) and if the inputs and the command line to create B.file are the same as the prior run, they don't rebuild B.file.
I tried to use makepp, but its lack of support for a pattern rule like:

```
report.%: database
```

where the prerequisite (on the right hand side of the ':') doesn't use any part of the base (%) made it unsuitable.
So a solution is provided in this diagram (the md5 mediated mode):

```
dist/A.file --- .sum/A.md5 --- A.report ---+
                                           +--- database
dist/B.file --- .sum/B.md5 --- B.report ---+
```

We add one more dependency between the per host report/cache file and the target file. This md5 file is generated automatically by the included Makefile.md5cache.include and placed in the .sum directory. Now when the database changes, file A.report and B.report are rebuilt. However, A.file is rebuilt only if A.md5 changes. The script that updates A.md5 calculates the md5 checksum of file A.report and compares it to the contents of the file A.md5. If they are the same, it exits and doesn't update A.md5 file. Since the A.md5 file's date hasn't changed, A.file isn't rebuilt. If the md5 sums are different it writes the new md5sum to the A.md5 file triggering a rebuild of A.file. In this scheme a change to the database does cause one update of every report file, but after that only the md5 update runs which is much faster and less resource intensive.

5.5.4 Performance of caching

The following example compares a direct dependency of the target files on the database to the md5 mediated dependency. Note that A.file and B.file also depend on a template file that is processed using the report files.
The following sequence is executed:

- the firewall configuration template is processed for every host (approx 90) to produce the output file.
- The database cache files are used by filepp to process the template

operation	*direct dependency*	*cache file*	*md5*	*%*
update database host A	1m47s	1m20s	1m17s	28
update template	1m45s	1m22s	29s	71
update with no changes	3s	1m28s	6s	-100

Table 5.1: Performance improvment using md5 mediated caching mechanism compared to direct mode.

We perform the following sequence of operations and record the wall clock time:
These were run with gnu make using 20 jobs in parallel (running serially the direct mode lasts for 7+ minutes).

- In 'update database host A' a change was made to the IP address of one of the 90 hosts. Then make was run.
- In 'update template' the firewall configuration template was changed. Then make was run.
- In 'update with no change' make was run and it has to run no commands.

The last column is the percent change between the direct dependency mode and the md5 mediated mode. All tests were run on the same set of hosts (approx 90). This is not meant to be a precise scientific experiment, the numbers above represent the best of three runs. It is also a real life example as the run was done on the box that is a master for DACS. So there is some jitter in the numbers due to the changing load on the box.
This table demonstrates the report/cache file generation dominating the processing time. The differences among the database update times is the first case is due to processing needed to generate the per host firewall files from the report and template files. So this provides the 28% time saving between the direct and md5 modes as only one firewall file had to be generated rather then all 90.
When we change just the template file, the md5 mode doesn't need to redo any reports, it just verifies the md5 files and starts processing. This provides the huge 71% performance increase. In the cache only case, we generate a report for every host except host A since the A.report file is newer than the database thus suppressing the report file generation. For the cache only case 23 seconds was saved from not having to regenerate the target files from the report files.
When there are no changes, make never has to run anything in the direct dependency mode. It looks at the timestamps and makes the determination. For the md5 mediated case we have 90 runs of the md5 comparison program which doubles the run time to 6 seconds for a whopping -100% 'improvement', but it's only 3 seconds. In the cache mode we once again are running multiple reports only to throw away the contents.

5.6 Using templates and other file processing tasks

Ok, so we know how to set up a Makefile to execute commands that can build files. Now why do we want this?

- It reduces the number of files that require hand editing. For example, the hosts file for every host has it's own name and IP address in it. At one site I worked at we also included the

DNS server and LDAP server information. The dns servers and ldap servers were defined in a separate file (the database) and the per host files were generated using this info. So to change a DNS server, required editing one file and the build system generated the 100 individual hosts files.

- It also reduces the change of errors from fatigue while editing all the files.
- The files generated from templates show less variation than manually maintained files. This makes developing programs to do changes to them, or audit them easier since there are fewer error conditions to consider.
- It reuses information in the database rather than embedding it into a configuration file. When the database is changed, the configuration file automatically gets the update.

Why should we not do this?

- It takes time to generate the files. If you only need three files, it may be faster to manually maintain them. Although you increase the risk of having out of date information in those files.
- The time spent implementing the file generation could be better used elsewhere.

However using the right tools and techniques within a framework can minimize both the build and implementation time.

5.6.1 Using filepp

While make allows you to use any processor to generate files, DACS provides support for generating files with filepp. Filepp is written in Perl and is meant as a generic file pre-processor providing:

- macro expansion
- foreach loops
- conditionals
- and other processing (including set manipulation)

It is freely available from `http://www.cabaret.demon.co.uk/filepp`.
If you have ever programmed in C or C++ you have used the cpp macro processor. Filepp input is similar to cpp, some of it's directives are identical to cpp.
Note that the distribution helper macros (see DacsDistribution) used by DACS need filepp, so it is likely that you will be exposed to it at least a little bit.
I cover the most common filepp directives you may see below. note that the directives in these examples are introduces with a '#' sign. This is settable on the filepp command line, but '#' is the default to agree with `cpp`. The directives shown below include the '#' in their descriptions.

5.6.1.1 Macros

A macro is simple a piece of text that is defined and replaced by another piece of text. Macros are case sensitive. The filepp directive `#define` is used to define a macro. For example:

```
#define MACRO REPLACEMENT
MACRO
```

will print REPLACEMENT anytime MACRO is seen in the file after the `#define` statement. Filepp also supports other type of macros (e.g. macros that take arguments) and ways of modifying macro values. I refer you to the filepp manual page for details.

5.6.1.2 Conditionals

Filepp also can select text to print using conditionals. The `#ifdef`, `#else` and `#endif` directives are used in:

```
#ifdef MACRO
macro is set
#else
macro is not set
#endif
```

will print `macro is set` if MACRO is defined (even if it is defined to be the empty value), and will print `macro is not set` otherwise.
In addition to testing to see if a macro is defined (or not defined using the `#ifndef` directive), filepp allows any Perl comparison operator to be used. So you can use regular expressions:

```
#if MACRO !~ /REPLACEMENT/
...
#endif
```

which will print `...` if MACRO expands to something that does not match the regular expression REPLACEMENT. While

```
#if "MACRO" eq "REPLACEMENT"
...
#endif
```

will print `...` only if MACRO expands to exactly the word REPLACEMENT.

5.6.1.3 Foreach loops

Using the foreach.pm module filepp obtains the `#foreach` directive and the `#foreachdelim` directive. This allows you to loop over elements of a macro. The elements are split apart by the value of the `#foreachdelim` (which by default is a comma, so in DACS it is usually set to one or more spaces). So the example:

```
#foreachdelim /\s+/
#define arguments a b, c d

#foreach THING arguments
the current element is THING
#endforeach
```

you will get:

```
the current element is a
the current element is b,
the current element is c
the current element is d
```

Note that the comma is included with the `b`. This is because the `#foreachdelim` was set to one or more white space characters, so the `b` and comma are part of the same white space delimited element.

5.6.1.4 Include statements

The `#include` directive locates a file and includes its contents into the current file.

```
#include "test.fpp"
```

will find the file "test.fpp" and insert its contents at the location of the include statement. The list of directories to search for include files can be changed on the filepp command line using the -I flag.

5.6.2 Basic directive/command examples

Let's take a look at the distribution file `firewalls/template.fpp`. In that file you see a series of filepp directives. Filepp directives/commands start with a # sign (although it can be changed if needed). Any line that starts with a # and is followed by something that is not a filepp directive is printed to standard output. If a non-directive line is inside a filepp directive like a conditional or foreach loop, it is processed by that directive. Other non-directive lines have any defined macros in the line replace and the result is printed to standard out.
This file is processed for each host using the database cache file for that host (Config/.host_cache_files/[hostname]),
The file starts with:

```
#pragma filepp UseModule foreach.pm
#foreachdelim /\s+/
```

A pragma changes how filepp behaves. This pragma makes filepp load the file that defines the foreach directive. Then one of the options for the foreach directive is set. This makes the foreach directive split its argument on white space rather than on commas.
Then we load a file of definitions.

```
#include "filepp.classes"
```

The filepp.classes is generated by DACS from the database and defines the various classes of machines like:

- FILEPP_SSHD_HOSTS - hosts running (the service) sshd
- FILEPP_CENTOS_4.X_HOSTS - hosts running some variant of the Centos 4 operating system

See the section on class definitions in the DacsDatabase chapter for details.
After this we have a few safety checks implemented using filepp conditional and error directives.

```
#if "LOCALNET" !~ /^[0-9.]+\/[0-9.]+$/
  #error LOCALNET not properly defined should be ip/netmask is 'LOCALNET'
#endif
```

The value for the LOCALNET macro (think of it like a variable) is the local network for the host being processed. If that variable doesn't match the regular expression of a series of digits and periods followed by a slash and some numbers (e.g. is not in the CIDR form 192.0.2.0/24), the `#error` filepp directive is executed and prints and error message and causes filepp to exit.
Further down in the file are the rules that determine if port 22 (the ssh port) should be made accessible. This is done by the (nested) conditional:

```
## ssh firewall rules
#if "SERVICES" =~ /\bSSHDEXT\b/
# allow all ssh traffic
-A IN -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
#else
  #if "SERVICES" =~ /\bSSHD\b/
# allow internal ssh traffic only
    #if "OS" =~ /FEDORA_2/
-A IN -s 192.168.0.0/255.255.0.0 -p tcp -m state --state NEW -m tcp --dport 22 -j ACCEPT
    #else
-A IN -s 192.168.0.0/255.255.0.0 -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
    #endif
  #endif
#endif
```

that emits the (iptables-save output) line:

```
-A IN -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
```

if the current definition of the SERVICES macro matches `SSHDEXT` standing alone as a word (so it won't match WSSHDEXT for example, `\b` matches a word boundary).
Otherwise it skips to the else part and if the word `SSHD` is found (again as a standalone word) in the SERVICES macro it will emit one of two lines depending on whether the OS matches FEDORA_2. If neither SSHD not SSHDEXT match, then nothing is printed from this nested conditional clause.
An example of using the foreach loop is provided by:

```
#foreach SNMP_IP SNMP_COLLECTOR_IPS
-A IN -s SNMP_IP -p udp -m udp --dport 161 -j ACCEPT
#endforeach
```

assuming the macro `SNMP_COLLECTOR_IPS` is defined with:

```
#define SNMP_COLLECTOR_IPS 192.168.1.1 192.168.3.1 192.168.5.1
```

the foreach loop prints:

```
-A IN -s 192.168.1.1 -p udp -m udp --dport 161 -j ACCEPT
-A IN -s 192.168.3.1 -p udp -m udp --dport 161 -j ACCEPT
-A IN -s 192.168.5.1 -p udp -m udp --dport 161 -j ACCEPT
```

and allows specific hosts the ability to query the snmpd daemon.
The IP addresses for the SNMP_COLLECTOR_IPS are extracted from the database using dbreport so adding a new service (e.g. nagios or cacti) to a host would automatically cause the firewalls on systems to change to allow it access to their snmp daemons.

5.6.3 Performing set operations with host classes

Filepp doesn't have any built in mechanism for dealing with sets (unlike the command used for file distribution), but we can use conditionals, foreach loops and an extra filepp directive or two to implement the standard 3 set operations:

- difference
- intersection
- union (straight addition)

and also unique union where each value in the set shows up just one.
All these examples assume that:

```
#pragma filepp UseModule foreach.pm
#pragma filepp UseModule predefine.pm
#pragma filepp UseModule predefplus.pm
#foreachdelim /\s+/
```

is prepended to each of the examples below. This is done by default using the `rdist_macros.fpp` file in `distfile.base` for example. Note that these operations works only if there are no spaces in the elements of the sets. However this is usually true in DACS as the set elements are hostnames which don't allow spaces
Note the spaces around the quoted macros in the if statements. They are needed to allow the macros (HOST, FILEPP_SET_B) to be replaced if `filepp` is run without the `-w` flag (substitute within words), it also means that you don't have to use `\b` in the regexp to specify a word delimiter as the space provides that functionality.
In these examples, the stuff in the inner loop are usually lines you want printed for each element that results from the set operation. If you want to define a new set, (that will be used 2 or more times in the file) an example using the `predefplus` directive is given in the "Difference" section.

5.6.3.1 Difference (FILEPP_SET_A - FILEPP_SET_B)

To process text for each host in FILEPP_SET_A that is NOT in FILEPP_SET_B use:

```
#define FILEPP_SET_A bar baz bax bam
#define FILEPP_SET_B baz bam

#foreach HOST FILEPP_SET_A
  #if " FILEPP_SET_B " !~ / HOST /
put stuff here that should occur for hosts/items in FILEPP_SET_A but
not in FILEPP_SET_B.
  #endif
#endforeach
```

This sets HOST to each host in FILEPP_SET_A. Then if FILEPP_SET_B does not matches the value of HOST, HOST is not in FILEPP_SET_B and we process and print the stuff between the `#if` and `#endif`.
To create a new set that is the difference of the two sets, use:

```
#define FILEPP_SET_A bar baz bax bam
#define FILEPP_SET_B baz bam foo

#foreach HOST FILEPP_SET_A
  #if " FILEPP_SET_B " !~ / HOST /
#predefplus DIFF_SETS_A_B HOST
  #endif
#endforeach

DIFF_SETS_A_B
```

will print `baz bam` and leave the new set defined as DIFF_SET_A_B for use later in the file.

5.6.3.2 Intersection (FILEPP_SET_A & FILEPP_SET_B)

To process text for each host in FILEPP_SET_A that is also in FILEPP_SET_B, we set it up like difference but change the if test from does not match ! to does match `~=<nop>`.

```
#define FILEPP_SET_A bar baz bax bam
#define FILEPP_SET_B baz bam

#foreach HOST FILEPP_SET_A
  #if " FILEPP_SET_B " =~ / HOST /
put stuff here that should occur for hosts/items in FILEPP_SET_A that
are also in FILEPP_SET_B.
  #endif
#endforeach
```

5.6.3.3 Union (FILEPP_SET_A + FILEPP_SET_B)

Unions are simple, just list the classes all on the foreach line

```
#define FILEPP_SET_A bar baz bax bam
#define FILEPP_SET_B baz bam

#foreach HOST FILEPP_SET_A FILEPP_SET_B
  put stuff here that should occur for hosts/items in FILEPP_SET_A and
  FILEPP_SET_B. Note that some things may be processed multiple times
  for the same host if the host is in both sets.
#endforeach
```

however if a host is duplicated in both FILEPP_SET_A and FILEPP_SET_B, it will be processed twice. To get around that a unique union must be used.

5.6.3.4 Unique Union (FILEPP_SET_A + FILEPP_SET_B) with duplicates removed

In some cases we have the same host in both sets (i.e. their intersection is not empty), but we want to emit something for that host only once. To do this we need a couple of extra filepp directives that were helpfully provided for me by the author, Darren Miller. From the email we exchanged:

> I've written a couple of new modules that replace macros when defined, so you can do what you wanted. The modules are included with this e-mail, just copy them to your modules directory.
>
> To do what you wanted, use #predefplus and a Perl regular expression if statement:
>
> ```
> #pragma filepp UseModule foreach.pm
> #pragma filepp UseModule predefplus.pm
> #foreachdelim /\s+/
>
> #define FOO bar baz bax bam
> #define BAR baz bam blah
>
> #foreach HOST FOO BAR
>   #if "DEF" !~ /\bHOST\b/
> HOST
> #predefplus DEF HOST
>   #endif
> #endforeach
> ```
>
> The two new modules provide the following keywords:
>
> ```
> #predefine macro defn
>   The  #predefine  macro  works  the  same  as the #define keyword
>   except any macros in defn are replaced at the time of the
>   definition, rather than waiting until the macro is used.
>
> #predefplus macro defn
>   The  #predefplus  macro  works  the same as the #defplus keyword
>   except any macros in defn are replaced at the time of the
>   definition, rather than waiting until the macro is used.
> ```

Note that the pragma's shown in the example are done automatically if you include rdist_macros.fpp, if you are doing a unique union operation in another context besides distfile.base, you will need the pragmas, or use -m and -M to filepp to load the modules and define the search path respectively.

Chapter 6

Distribution of files

Distribution is done using the script (capital R) Rdist which uses the rdist(1) (little r) command over ssh to distribute files and run commands.
The rdist(1) command copies files preserving the:

- owner
- group
- mode
- and contents

of the master copy to a target copy residing on another machine. It uses a mechanism such as ssh(1) to gain access to the remote machines.

6.1 What does the distribution system provide

The distribution mechanism in DACS supports:

- updating files on remote systems based on file contents or modification times.
- running commands remotely when a file changes
- backing up a user definable number of prior copies of a updated file. This makes it possible to undo changes or compare prior and current files on a system without having to go through DACS. This speeds problem diagnosis and resolution when investigating a problem after an update.
  - the files can be backed up to a different directory from the installation directory. This is used when pushing files to a config directory (e.g. /etc/cron.d), where any file in the directory is considered a command file. By moving the backup to a different directory you prevent the prior copy of the file from being used.
- three types of distribution commands:

  - pre commands - runs before a file is distributed. It can be used to set up the environment to receive the distributed file, determines the state of dynamically changeable services etc.
  - installation commands - installs a file and runs commands (if needed).
  - post commands - runs after an installation command is run to verify state changes, perform cleanup, notify of pending manual actions etc.

- grouped post commands - if you have 5 files to distribute and changing any one of them requires a restart a program. This allows you to only do one restart even if all 5 files are updated (which normally would result in 5 restarts of the program).

- forcing a command to run every time it is invoked. By default commands are only run when a file is updated.

- report the file destination (on the remote machine) in addition to the source file name.

- verification targets that run when Rdist is in verify mode and allows programs to run that can verify state. Normal verification only reports file differences, this can perform arbitrary commands to determine if things are in a sane state.

- reporting differences between two files (useful for verification targets that report what will change before you distribute files)

- logging of file updates to systems

There are approximately 10 filepp macros that are used to provide some of this functionality and hide the ugly details. Their use is documented (Config/docs/lib/config/rdist_macros_fpp.html) and examples are provided in this documentation and the distribution example files.
DACS expects to use version 6 of the rdist command. There were some major advancements in that release of rdist:

- rdist can distribute to multiple systems in parallel

- rdist can do set operations on space delimited lists of hosts

- rdist can run over ssh rather than rsh

All of these greatly increase the functionality or security of rdist. Version 6 of rdist (released in 1992) is the version of rdist that is distributed by most OS vendors. If your OS doesn't support it, the source code is available and it can be built and installed. It has support for using the native rdist binary if it is invoked with the older style command line so it provides backward compatibility.
To set up rdist access and allow Rdist to work, see "Setting up ssh access from the master to clients" section 9.2.

6.2 Running Rdist

The main command used under DACS is Rdist which is at the top of every checked out configuration tree. An sample invocation is: `sudo /config/Rdist label1 label2 ....` where the `label1...` arguments specify parts of the CCM tree to update and distribute.
Using labels reduces processing and distribution time as well as limiting the changes that will be pushed. Since there can be multiple administrators working in the CCM tree at one time this allows you to limit your distribution to the area you changed. Rdist uses the rules contained in the file `Distfile`, that is located at the top of the DACS tree, to determine what files are distributed and what commands are run on remote hosts.
The Rdist script does the following on every invocation:

- Locks the CCM tree from simultaneous updates.
- Updates the DACS mechanisms under the Config subdirectory (including itself) from the VCS.
- Generates a Distfile from Config/distfile/distfile.base and the DACS database in Config/database/db. This adds variables to the Distfile that contain hostnames meeting some criteria. It also expands any DACS macros/syntax in distfile.base into standard rdist(1) command stanzas.
- Runs a host verify step that verifies information in the database (host name, os type and version, architecture ...) against the host. This makes sure that dns mappings are correct and that the host that Rdist thinks it's talking to is the host defined in the database.
- Determines the list of labels that are to be distributed. The label list is either given on the command line or is determined by scanning for the the automatic labels (defined below) in the Distfile.
- Determines the list of machines to be updated. These are either given on the command line with the `-m machine.name` flag or excluded from the default list using `--exclude machine.name`. The default list is generated from the DACS database by looking for machines that have the `rdist` keyword set to 'yes'.
- Does an VCS update of the directory corresponding to the labels it's processing to get new files and modifications.
- Invokes the build mechanism for each directory corresponding to a label it is processing if the directory has a Makefile in it.
- Invokes rdist(1) using the machine list and labels determined above to distribute the files.

If the host verify step fails for a host, that host (and only that host) is removed from the list of hosts that will be updated. The distribution continues with any verified hosts. Host verification can be disabled using the `--nohostverify` option to Rdist.
If any excluded host specified with `--exclude` is not found in the DACS database, the distribution is stopped. However if a host specified with `-m` is not found the distribution continues but without that host. The theory is that you specify `--exclude` to stop an undesired state

change. If the excluded host is not found and the distribution continues, the host that you didn't want to change would be changed. This is bad since it may be difficult to roll back the change without rolling back the entire state of the repository. On the other hand if you mistyped a host to the `-m` flag, no change is done to the host and it can be easily fixed (in most cases) by running another Rdist with the proper host name.
If any of the VCS updates fail because:

- the VCS server is unavailable
- the update finds files that aren't supposed to be present and aren't marked as ignorable
- the update discovers any changed files in the tree (don't forget the DACS distribution tree should consist only of exact copies of files in the repository and files derived from them. Otherwise you can't roll back the configuration.)

the distribution is stopped. If you have directories such as dist or .sum (discussed below) that contain artifacts from the build systems, you need to tell the VCS to ignore the files in these directories. See "VCS Interaction" section 5.5.1 in the DacsBuild chapter.
Also if any of the build/make operations exit with an error condition, the distribution is stopped. These safety interlocks help prevent pushing incorrect, incomplete or invalid files to machines. There are a few more optional things that are done, see the online Rdist documentation (obtained by running `perldoc` on the Rdist script) for details, but this is sufficient for our discussion.
Rdist can operate in two modes: distribute and verify. In distribute mode the default labels are the automatic labels. In verify mode, (invoked with `Rdist -v`) the default labels are the automatic and verify labels (see below for information on the definition of these labels). In both cases the labels are sorted alphabetically when used. So the commands associated with the `etc` label will occur before the commands associated with the `tomcat` label.

6.2.1 Other Rdist options

For a full list of options to Rdist, use `perldoc Rdist`. The options that are used in this documentation are listed below.

-v	verify files only. Do not push any operational files.
-S c	generate a condensed summary of rdist output
-S v	generate a verbose summary of rdist output

Table 6.1: Common arguments to Rdist command

The condensed summary reports all hosts that receive a particular file. So it groups by distributed file. E.G.

```
s5.example.com s7.example.com
    /config/etc/hosts: need to update
    /config/sshd/known_hosts: need to update
```

On the other hand, the verbose summary groups all the output for a host together.

```
s5.example.com: /config/etc/hosts: need to update
s5.example.com: /config/sshd/known_hosts: need to update

s7.example.com: /config/etc/hosts: need to update
s7.example.com: /config/sshd/known_hosts: need to update
```

They are useful to make sense of the Rdist output when multiple hosts are updated in parallel.

6.2.2 Host selection and verification

If no host is specified, each host in the database with the `rdist` keyword set to `yes` is selected as a client. A host or hosts can be specified on the Rdist command line with the `-m` flag. The argument to the `-m` option can be a single host name (as specified in the database using the machine keyword) or a space separated list of hostnames again matching the `machine` keywords in the database. The space separated form is useful for selecting a group of hosts meeting some criteria (e.g. they all run NTP) from the database using `dbreport -l`.
If you don't use the `-m` flag, you can exclude hosts from the default set using the `--exclude` option. The argument to `--exclude` has the same formats as `-m`. At some point in the future you should be able to specify both `-m` and `--exclude` and have the right thing(TM) happen.
If the HOST-VERIFY option is not disabled using `--nohostverify` then the host verify script is executed on each client. If the script reports an error, the host is removed from the machine list. Currently the script checks that:

- the host's name (returned by hostname) matches the one in the database
- the host is running the same os release as is specified in the database
- the host has the same architecture as specified in the database. (If no architecture is specified in the database, this check is skipped.)

This reduces the chance of pushing incorrect files to hosts.
The need for this arose when a Sun workstation was swapped for an Silicon Graphics workstation. They had the same name and IP address, and the database entries weren't updated. Needless to say many new words and sounds were invented when the Silicon Graphics system was updated.

6.3 Target Overrides and Rdist fragments

This is here for historic documentation. At this time I don't know of anybody who uses this, but people transitioning from some private releases of Config may use it. If you do need/use this please let me know and we can try to work out a better defined replacement. If you are just getting started with DACS, skip this section totally as it scares me and I am sure it will will just scare you. It has not been tested in a long while and may not work. I look forward to removing it.
When you specify a target to Rdist a number of things happen if an associated directory exists at the root of the tree:

1. The directory is locked against other Rdist updates by creating a .locked directory with a file locker containing the username@host and process id (pid) info. The lock directory is recorded to allow deletion after Rdist exits.

2. The directory is updated from SVN.

3. If a Makefile exists in the directory, "make .targets" is executed if run in distribution mode, or "make .targets-verify" in verify mode. Note that .targets-verify is a phony target and the file should not be made. The verify target should be updated.

4. If a `.targets` file exists, the specified target is replaced by the contents of the .target file with a $$ in the `.target` file replaced by the original target. This can be used with pre/post targets to explicitly order a sequence of steps. Path targets can be used to map the target into a deep directory structure. Also the file can be empty if you use a .push file to specify an alternate distribution mechanism.

5. for each element of the target list (if a .target was found), it is updated from SVN if the target exists in the file system.

6. if a Makefile exists in the directory, make is run with no arguments in distribution mode, or a `verify` target in verify mode. Before running the verify target, the `Makefile` is scanned for the verify target to prevent it from running is there is nothing to do. If make exits with an error, the Rdist is stopped in distribute mode, but is ignored in verify mode.

7. if the .targets file was empty and a .push file exists, it is queued up for future execution.

8. if a file .rdist exists, it is appended to the master distfile. Note that targets defined in the .rdist file will not be automatically found, so a .targets file must exists if a new target is defined in that .rdist file.

6.4 Controlling Rdist: the Distfile

This section discusses Distfile structure and conventions including filepp macros used by DACS to simplify the specification.

6.4.1 The anatomy of a Distfile entry

As mentioned above, the Distfile is used to drive the Rdist/rdist mechanism. A typical Distfile entry stanza looks like:

```
label:
source/file(s)/or/directories -> ( host(s) )
            install -ooptions target/file_or_directory_location/on/host;
            cmdspecial "shell command run on remote system if something changes";
```

In DACS, filepp macros have been defined to make some of the features work more easily and we will discuss those shortly, but let's look at this example. Since Rdist always specifies labels to use for distribution, each distfile stanza should start with a label. There are a number of different label types which are discussed below, but all are on a line by themselves and end with a ':'.
The next line in the stanza specifies the source file or files that are to be distributed. You can use shell style glob specifications (e.g. `*` or `{a.file,b.file}`) to specify more than one file. If you are installing a directory of files, you will want to use the `except` or `except_pat` commands to

prevent certain files (like the subversion of CVS control directories) from being pushed. See the "Distfile.base Examples" section 6.4.5 below or rdist(1) manual page for details.
After the '->' token is a list of hosts. Usually in DACS it is not a literal list of hosts like `(a.example.com b.example.com)` but an rdist variable that lists a class of hosts (hereafter called a class or class variable) sharing some attribute. Some sample classes are:

NTPCLIENT_SLAVES a list of hosts in the DACS database that have the line `uses=NTPCLIENT ....` in their definition.

SSHD_HOSTS a list of hosts running the ssh (secure shell) daemon that have `service=SSHD ...` in their definition.

These classes are automatically generated by the Rdist command using dbreport and the DACS database and are available for use.
After the source and host specification is the indented command `install` that installs the files before the `->` token at specified location. If there is only one file being installed, the destination location can be a directory or a file. If multiple files are to be installed, the target must be a directory. There are various options that can be specified that will:

- perform a binary file compare rather then just checking the modification time of the file
- create a backup of the original file before it's overwritten
- ignore owner, mode or group differences
- for all the options see the rdist(1) man page.

The rdist command `cmdspecial` is optional. If it is present the shell command specified is executed if the install command caused any file to be updated. If no files were updated the cmdspecial doesn't do anything. The shell command associated with the cmdspecial is run after all the files are installed. There is a `special` command as well that is run once for each updated file if you are installing multiple files.
If you look at a generated Distfile, you will see many of these commands used.
However you don't modify the Distfile. You place your commands in the file Config/distfile/distfile.base. Because it can be tedious setting up many of these commands DACS provides macros like `SAVEINSTALL` that expands to multiple basic rdist commands to implement specific functionality. The macros are described below, but first a discussion of the label types is appropriate since that is how all the rdist stanzas start.

6.4.2 Distfile Label Types and Special Purpose Labels

As mentioned above, there are a few different label types. They are explained in detail in this section. Note that some versions of DACS/Config documentation refer to these as targets. Labels and targets are the same in the context of rdist. These label types are purely a DACS convention and don't exist as far as rdist(1) is concerned.
The rdist stanza types are defined by their labels. So if a stanza is labeled with an automatic label, it is considered an automatic rdist stanza or rule. The 6 basic label types (you will only care about the first 4 unless you are converting from Config) are:

***automatic* Specification** Automatic labels start with a lower case letter and consist of letters (upper/lower case), numbers and underscores.

Purpose these targets are used by default when Rdist is invoked without any explicit labels. So they are automatically run if no labels are specified. They are also sometimes referred to as normal labels.

Associated Directory the name of the automatic label is expected to be a top level directory in the DACS tree. This directory will be updated from the VCS automatically and it will be searched for a Makefile to trigger a build process.

Notes none

***verify* Specification** Verify labels are automatic labels that end in `-verify`.

Purpose Verify labels are used in addition to automatic labels when `Rdist -v` (verify mode) is run with no labels. The rdist(1) verify mode just compares the files in the repository against the files on the host. It does not execute any commands and does not change any files. In some cases this is insufficient to actually verify that things are up to date.

For example a managed file may need local post processing to be activated/installed, so you have to undo the post-processing to see if the proper rules are in place. Rdist running in verify mode runs the '-verify' labels in distribution (not verify) mode using rdist(1). This allows file updates and commands to execute. Hence it can push files and call scripts. It can verify that an installed firewall rule set matches the spec file for the firewall rules. It can also run monitoring programs although a tool like nagios would be better suited for active monitoring.

Associated Directory The directory update operations use the name of the label without the '-verify' suffix. So a "firewall-verify" label would update/build the `firewalls` top level directory. When the build occurs in verify mode, the makefile command "make ... verify" is used so you can build different things in a verify compared to a distribution run.

Notes Verify labels should not make operational changes to the remote system. This is not enforced but unexpected and bad things can happen if this rule is violated.

***partial* Specification** Partial labels have a period '.' in their names.

Purpose Partial labels are meant to allow pushing a subset of the files under a directory. E.G. you may have an `ntp` target, but you can also define a target (`ntp.toplevel`) that pushes configuration files for only the top level ntp hosts. Or you can have a target `etc.sudoers` that pushes only the sudoers file among all the files under the etc directory.

It is expected that stanza's labeled with a partial label are also duplicated with an automatic label so that the files are automatically maintained.

Associated Directory The directory update operations use the name of the label preceding the first period '.'. So a "etc.sudoers" label would update/build in the `etc` top level directory.

Notes A partial target is an automatic target with a period and trailing text. So you can not create a partial `path` label.

***manual* Specification** Manual targets have `-manual` appended to them.

Purpose Manual targets are never automatically run and must be specified manually (hence the name) on the Rdist command line in order to be pushed. These are useful for one time changes (e.g. commands to run only on initial system setup) or other temporary changes (e.g. pushing an alternate configuration that blocks Internet access) that need to be imposed.
The files managed by a manual label may be but are not expected to be managed by any other label

Associated Directory The directory update operations use the directory preceding the -manual suffix. So "postgres_upgrade-manual" would update the postgres_upgrade directory.

Notes none

path Notes This is kind of useless in it's current state unless you are using target overrides or distfile fragments. Best you forget about it.

Specification Uses forward slashes '/' between components

Purpose path targets are used to bridge the gap between a deep directory structure and a shallow target name space. By default target names are expected to map to directories that will have various operations performed on them. Path targets are just regular targets with '/' characters separating to components. Note they do not start with a /. Verify and other types also apply to path targets, but they are not automatically selected and must be explicitly specified.

Associated Directory: None. No build or VCS updates are done for these items.

distribute Notes Consider this deprecated. However if you are using it from Config let me know and I will add support for it.

Specification Distribution targets have -dist appended to them.

Purpose Distribute targets are invoked only in distribution mode and not in verify mode. Note that this mode is not yet implemented in DACS Rdist although it did work in Config's Rdist. I don't think I ever got any reports of people using these, but a possible use case would be with a firewalls update where the stanzas associated with the '-verify' label are the only useful operation and the verification provided by running the automatic label in rdist's verify mode is useless. So you could create a firewalls-dist rule that runs only when distributing files and never when verifying to save time.

Associated Directory The directory update operations use the directory preceding the -dist suffix.

There are also two additional types of labels that usually don't require special specification. By default rdist has an implicit execution order based on the order of the distribution stanzas in the distfile. So for stanzas with the same label, their distfile specification order determines the execution order. This mode is supported and should be used.
However DACS supports the ability to create the distfile from fragments located in the DACS tree. It also supports the ability for one label to add other labels to the distribution list. This supports a very distributed mechanism of update and delegation where multiple admins from

different groups are responsible for updating specific trees and the main control files are not editable by all of them. This support may be removed in the future as I have decided I don't like the model or implementation. However I am documenting it here.
Each type below has two description paragraphs, the first paragraph describes the use case for the implicit execution order.
You should skip the second and subsequent paragraphs for each type unless you are coming from the Config system where it is used. The two additional label types are the pre and post labels that are expected to run before (pre) and after (post) some other stanza.

pre a pre label is any one of the 4 standard label types that is run before another stanza. It can set up conditions for the other stanza to work (e.g. by updating/pushing commands used by a cmdspecial, or by creating a file needed by another stanza). Basically it is a helper to a stanza that actually does something useful. It has no special markup and shares the same label as it's associated command. However it must occur before the associated command in the distfile so that it runs prior to the associated command. See "Configuring dynamically reconfigurable services" section 7.10 in DacsExamples for an example of pre label use.
There is a second variant of this that is useful in very complex environment where the Distfile is built in pieces and allows explicit execution order independent of the order of the stanzas in the distfile. If you need this read on otherwise stop now (unless your health plan has very good psychiatric services). If you use the target override methods and rdist fragments it is possible to explicitly specify the execution order. You can't use the implicit order of the stanzas in the distfile as the rdist fragments are appended to the resulting Distfile. In this case, the pre label is an automatic label followed with '-pre'. Numbers may be appended to the `-pre` suffix to permit multiple ordering levels independent of the order in the distfile (e.g. `-pre1`, `-pre2` etc.). No automatic directory update or builds occur with a pre target. (Slight lie, the suffix isn't stripped so a directory of mumble-pre would have to exist to be updated, but this is probably not a useful function.)

post a post label is exactly like the pre label except that it occurs after it's associated target. It can verify correct update, send notification emails etc. They aren't used much since most of this can be done with special and cmdspecial directives in the associated stanza, but it may be useful in some cases (e.g. such as managing network devices via proxy, or always running a command even if no files were updated (which prevents the cmdspecial and special command in an automatic label from running)).
Again this is a place you probably don't want to be reading. But if you are among the sites using rdist fragments or target/label overrides a post label is identified by the suffix `-post` on the label name. As with pre targets, numbers may be appended to the `-post` suffix to permit multiple ordering levels. No automatic directory update occurs with a post target. (Slight lie, the suffix isn't stripped so a directory of mumble-post would have to exist to be updated.)

There are also two special labels that are used internally by the DACS mechanisms:

"HOST-VERIFY" is a special label that is run to verify basic facts about a host. If the remote host fails to verify, Rdist removes the host from the list of hosts to update.

"POSTINSTALL-ACTIONS" is a special label that is at the end of the generated distfile so it executes after all other targets. It is used to execute any final commands that need to be done. It is primarily meant to trigger the restart of programs like httpd when any one of it's configuration files changes. The example "Triggering one service restart for multiple files" below describes it's use.

6.4.3 Distfile Macros

The macros that ae used in Distfile creation are documented in the file `Config/lib/config/rdist_macros.fpp`. This file is included in the default `distfile.base.example` that is distributed. The macros will be explained in the "Distfile Examples" section as they are used, but I suggest you look at the documentation generated from the source. This documentation is located in the distribution at `Config/docs/lib/config/rdist_macros_fpp.html`. The key macros and their synopsis are:

RDIST_SPECIAL_FILTER	use with special commands to filter it from the output
BACKUP	manage some number of backup copies of pushed files
BACKUPTO	backup copies of pushed files to a new directory
DATESTAMP	a current datestamp
FORCERUN	always run special commands
NOOP	Allow a rule in distfile that will never be executed
IFPOSTACTION	check to see if there are postinstall commands to run
POSTACTION	register an action for execution after all updates occur
REPORTTARGET	report the name of the updated file as it is known on the target machine.
SAVEINSTALL	Install a file saving some number of copies.
VERIFYFILE	Diff an installed file against the one in the repository.

Table 6.2: Macros defined for use with Rdist in distfile.base

Note that the macro names are all uppercase to make them stand out from normal rdist commands.

6.4.4 Distfile set operations

The version 6 rdist command includes the ability to perform set operations between two class variables. Three operators are available:

Set Operation	*Operator*
difference - in set A but not set B	-
intersection - in both set A and B	&
union - in set A or B	+

Table 6.3: Set operations supported by rdist.

For example:

```
NON_CENTOS_HOSTS=${ALL_HOSTS} - ${CENTOS_HOSTS}
```

```
JUST_CENTOS_HOSTS=${ALL_HOSTS} & ${CENTOS_HOSTS}

JUST_SOLARIS_AND_CENTOS_HOSTS=${SOLARIS_HOSTS} + ${CENTOS_HOSTS}
```

The rdist command doesn't support more than two variables. When more complex set combinations are needed, intermediate variables have to be used:

```
ONLY_CENTOS_4_1237_HOSTS_1= ${CENTOS_4.1_HOSTS} + ${CENTOS_4.2_HOSTS}
ONLY_CENTOS_4_1237_HOSTS_2= ${CENTOS_4.3_HOSTS} + ${CENTOS_4.7_HOSTS}
ONLY_CENTOS_4_1237_HOSTS = ${ONLY_CENTOS_4_1237_HOSTS_1} + ${ONLY_CENTOS_4_1237_HOSTS_2}
```

or

```
ONLY_CENTOS_4_1237_HOSTS_1= ${CENTOS_4.1_HOSTS} + ${CENTOS_4.2_HOSTS}
ONLY_CENTOS_4_1237_HOSTS_2= ${ONLY_CENTOS_4_1237_HOSTS_1} + ${CENTOS_4.3_HOSTS}
ONLY_CENTOS_4_1237_HOSTS =  ${ONLY_CENTOS_4_1237_HOSTS_2} + ${CENTOS_4.7_HOSTS}
```

will create a list of hosts running centos 4.1, 4.2, 4.3 and 4.7.

6.4.5 Distfile.base Examples

In DACS you don't edit the Distfile directly. The Distfile is generated from distfile.base and the DACS database by make with the assistance of filepp and dbreport. This section displays some examples from distfile.base using the macros and other filepp magic that makes specifying the distfile rules easier.
In all these examples you will see that the source file always begin with `$C`. The variable `C` is defined by the Rdist command when it calls rdist and always points to the top of the current configuration tree. This allows you to have multiple configuration trees checked out without having to modify the distfile rules.
So invoking `sudo /config/Rdist ...` will set `$C` to `/config`. while invoking `sudo /config.test/Rdist ...` will set `$C` to `/config.test`.
See `distfile.base.examples` (and `distfile.base.config.examples`) distributed with DACS for more examples. These examples are intended to give you a feel for what can be done and cover the usual cases.

> Note: the command lines in the examples below may be too long to display. So they are wrapped using a \ at the end of the prior line. In distfile.base these continued lines must all be on a single line and the \ indicating the configuration must be removed.

6.4.5.1 Establishing a standard time on all systems

How many times have you had to look at date-stamped log files and realized that the date-stamps are from different timezones. To make all your timezones consistent you can push the same timezone file to all your hosts. This rule does just that pushing etc/localtime/UTC.linux as /etc/localtime.

```
# the localtime file can not be a link into /usr/share/zoneinfo because
# timezone is needed when system is booting before the /usr partition
# is mounted.
```

```
etc:
$C/etc/localtime/UTC.linux -> ${TZUTC_SLAVES} & ${LINUX_HOSTS}
        SAVEINSTALL(/etc/localtime,2);
```

It performs an intersection of the host classes that use TZUTC (i.e. have `uses=... TZUTC ...` in their database definition) and hosts that run LINUX. So if you have a Solaris host or a BSD host running with TZUTC they aren't updated by this rule.
Now there is on wrinkle to this. You might think that the LINUX_HOSTS have `os=LINUX`. Although this is logical, it's also wrong in this case. If you run multiple Linux distributions: SuSE, Centos, Debian, Fedora Core, ... you want to specify each distribution as the OS since at an administrative level they behave as different operating systems. You could add a LINUX uses value, but when you bring up a new host, the fewer settings you need the better. So the way LINUX_HOSTS is defined is below:

```
LINUX_HOSTS=${FEDORA_HOSTS} + ${CENTOS_HOSTS}
```

That is it's the union (indicated by the + sign for rdist) of the two automatically generated specific OS classes. Let's say that we also ran system with the Debian release. To generate a list of all Linux hosts in this case you would need to use:

```
LINUX_HOSTS1=${FEDORA_HOSTS} + ${CENTOS_HOSTS}
LINUX_HOSTS=${LINUX_HOSTS1} + ${DEBIAN_HOSTS}
```

you can only connect two variables with the + (union), & (intersection) or - (difference) operators. So you need to create numbered variables to hold the intermediate results. Sadly this:

```
LINUX_HOSTS=${FEDORA_HOSTS} + ${CENTOS_HOSTS} + ${DEBIAN_HOSTS}
```

will result in an error from rdist.
We also see the new macro SAVEINSTALL. It takes three arguments:

1. the destination location for the file(s) on the host
2. the number of backups copies of prior versions of the file(s) to keep
3. any options to pass to the install command (e.g. compare). The third parameter is optional and is missing in this example.

6.4.5.2 Pushing unique /etc/hosts files for each host

This example expects the build system to create unique hosts files for each system under `etc/hosts/dist/<hostname>` where =¡hostname=¿ is replaced with the machine name specified in the database.
This stanza pushes the hosts files generated from the build system to all the client hosts running FEDORA core or CENTOS that are not listed in the (manually maintained) OLD_HOSTS rdist variable (see Supporting Legacy hosts... section 6.5 for why you shouldn't do this).
It uses filepp directives to duplicate a template once for each host. (See Using Filepp section 5.6.1 for information on using filepp and it's directives.)

```
#foreach HOST FILEPP_CENTOS_HOSTS FILEPP_FEDORA_HOSTS
etc:
$C/etc/hosts/dist/HOST -> ( HOST ) - ${OLD_HOSTS}
        install -osavetargets,compare /etc/hosts ;
        BACKUP(10);
#endforeach
```

The macros FILEPP_CENTOS_HOSTS and FILEPP_FEDORA_HOSTS are automatically generated by DACS from the `os` keyword values in the database.
The filepp foreach loop emits a list of stanzas that look like:

```
etc:
$C/etc/hosts/dist/a.example.com -> ( a.example.com ) - ${OLD_HOSTS}
        install -osavetargets,compare /etc/hosts ;
        BACKUP(10);
```

and so on for b.example.com, c.example.com and the rest of the hosts in the FILEPP_CENTOS_HOST and FILEPP_FEDORA_HOSTS macros. So each host specific generated hosts file is pushed to it's corresponding host. The old copy is saved (-osavetargets) and the install command uses a binary comparison to make sure the file is identical and to prevent pushing a file when only the datestamp of the generated file changes.
Since the source file changes for each host, we have to use filepp to generate the list of stanza. Well we don't HAVE to use filepp. We could write them all manually, but I hope you will agree that using filepp is easier and less prone to error.
The BACKUP(10) macro saves the 10 prior installed versions of /etc/hosts (except for the most recent) on the client named: `/etc/hosts.SAVED` which is the most recent saved file and prior versions `/etc/hosts.SAVED.~N~` where there will be 10 files with `.~N~` suffixes and the N is the N'th version of the replaced file. E.G. if you have the files: `/etc/hosts.SAVED.~14~` ... `/etc/hosts.SAVED.~23~` you know that the current /etc/hosts is the 25th version to have been pushed (as `/etc/hosts.SAVED.~23~` was the 23rd version, /etc/hosts.SAVED is the 24th version and /etc/hosts is the 25th version). You won't see `/etc/hosts.SAVED.~13~` because only 10 prior `~N~` files are preserved. There was at one point a `/etc/hosts.SAVED.~1~` file, but it has expired.
This could also be written as:

```
#foreach HOST FILEPP_CENTOS_HOSTS FILEPP_FEDORA_HOSTS
etc:
$C/etc/hosts/dist/HOST -> ( HOST ) - ${OLD_HOSTS}
        SAVEINSTALL(/etc/hosts, 10 , compare );
#endforeach
```

and would have exactly the same effect (in fact both versions expand to the same result).
On the other hand if you have a single hosts file that is valid for every Linux host in your organization you can use the simpler:

```
etc:
$C/etc/hosts/combined_hosts_file -> ${LINUX_HOSTS} - ${OLD_HOSTS}
        install -osavetargets,compare /etc/hosts ;
        BACKUP(10);
```

Where LINUX_HOSTS is defined as above.

6.4.5.3 Pushing a daily cron job to backup mysql

The following stanzas install a cron job that backs up a mysql database daily. It is installed on all hosts running mysql as listed in the rdist class variable `MYSQL_HOSTS`. It also shows the use of a verify stanza/label.

```
# Backup mysql databases on all hosts running mysql.
cron:
$C/cron/cron.d/mysql_dump -> ${MYSQL_HOSTS}
        SAVEINSTALL(/etc/cron.d/mysql_dump, 2, compare) ;
        cmdspecial "if ! test -e /var/bak/mysql.all_databases.dump; \
             then touch /var/bak/mysql.all_databases.dump; fi \
             RDIST_SPECIAL_FILTER";
        BACKUPTO(/etc/cron.d.backup, 5);

cron-verify:
$C/cron/cron.d/mysql_dump -> ${ALL_HOSTS} - ${MYSQL_HOSTS}
        FORCERUN;
        cmdspecial "if [ -r /etc/cron.d/mysql_dump ]; then echo \
        \"ERROR: /etc/cron.d/mysql_dump exists on a machine that is \
        not a MYSQL server. Please remove.\"; fi \
        RDIST_SPECIAL_FILTER";
```

When you run `Rdist cron` it will:

- push the file cron/cron.d/mysql_dump to
- all hosts in the MYSQL_HOSTS class (a host with `service ... MYSQL ...=` in it's definition)
- install the file (if the target is different) as /etc/cron.d/mysql_dump and create a backup
- it will compare the file and not just use the date of the file
- if the file is updated, it will make sure the directory used by the cron job exists and creates it if it does not exist. The command will not be displayed to the user by default.
- the backup file will be moved to the /etc/cron.d.backup and 5 copies of the file will be kept.

If you choose the label cron-verify (or run `sudo /.../Rdist -v` which will include the -verify labels automatically) it will:

- for the set of hosts in ALL_HOSTS that are not in MYSQL_HOSTS (i.e. it does a difference of the two sets).
- push the file cron/cron.d/mysql_dump to some random location. This will always succeed in updating the file which will trigger:
- a command that verifies that there is no `/etc/cron.d/mysql_dump` file on the machine.
- It will complain if the `/etc/cron.d/mysql_dump` file is found since mysql isn't supposed to be running on that host.

Now you may ask how did the `mysql_dump` file get there in the first place? Well the host may have run mysql at a prior point and it was disabled and the mysql_dump file was never cleaned up.
Before we discuss the new macros, note the use of `\"` to embed double quotes inside the cmdspecial. If the double quotes are not escaped using the backslash `'\'` they terminate the cmdspecial (or special) command and will cause a syntax error.
The RDIST_SPECIAL_FILTER macro takes no arguments and sits at the end of any special or cmdspecial command inside the quotes. Note that there is no ';' between the shell command and the macro. All it does is supply a unique string that will be filtered out by Rdist when run unless you specify a high verbosity level. This simplifies the output making it more readable.
You can do this to remove housekeeping commands from the Rdist report. Note that you probably don't want to do this for commands that are needed to restart/reload services for two reasons:

- errors from those commands can be confusing if the command is not shown
- if the operator needs to perform the same operation by hand, it is easy to see what commands should be run to update/reload the service.

Housekeeping commands are part of the structure needed to allow rdist to manipulate the service configuration as opposed to the commands that actually change the service configuration. So:

- don't filter "/etc/init.d/httpd graceful"
- do filter "cp /etc/http/conf.d/proxy1.cfg /etc/http/conf.d/proxy1.cfg.bak"

The BACKUPTO macro moves the backups done by SAVEINSTALL (or the basic rdist command `install -osavetarget`) to a different location. It takes two arguments:

1. the directory the backup file should be moved to. If the directory doesn't exist it is created mode 700 owned by root.
2. the number of backups to keep.

When using both BACKUPTO and SAVEINSTALL, the number of retained backups is determined by the BACKUPTO macro.
Why are we using the BACKUPTO macro? Well cron loads all the files in the directory `/etc/cron.d`. If we left the backup file in the same directory, which is the default and is reasonable in most cases, the backup file would get loaded as well. So we use BACKUPTO to remove the backup and place it in a different directory hiding it from cron.
The last new macro is FORCERUN. All this does is make sure that a file is updated and it triggers any cmdspecial or special commands. It has no other function other than to force the running of special or cmdspecial commands.

6.4.5.4 Excluding files in a directory from being pushed

In this example I am pushing all the files in the directory `config/dist` but I don't want the administrative files pushed to the clients. So I exclude the directories: `.svn`, `.locked` and the files `Makefile` and `empty` from the list of distributed files.

```
# install files needed to support DACS
config:
$C/config/dist -> ${ALL_HOSTS}
        install /etc/config/. ;
        except_pat ( \\.svn ) ;
        except_pat ( Makefile ) ;
        except_pat ( empty ) ;
        except_pat ( \\.locked ) ;
        REPORTTARGET;
```

The except_pat command takes a regular expression. The double back slashes before the period in the `.svn` and `.locked` entries make the period match only a period and not any character. Because backslash `'\'` is an escape character is must be doubled when used to represent itself.

6.4.5.5 Duplicating stanza's with automatic and partial labels

One of the drawbacks of partial labels is that you need to duplicate the same rule with an automatic label. However since filepp is used to generate the Distfile from distfile.base you can use this construct:

```
#foreach LABEL users users.sudo
LABEL:
$C/etc/sudoers/sudoers.site -> ${LINUX_HOSTS}
        SAVEINSTALL(/etc/sudoers, 3, nochkmode);
        special "chmod 440 /etc/sudoers";

LABEL:
$C/etc/sudoers/sudoers.site -> ${SOLARIS_HOSTS}
        SAVEINSTALL(/etc/sudoers, 3, nochkmode);
        special "chmod 440 /etc/sudoers";
#endforeach
```

which will generate two copies of these stanzas. One copy with `LABEL` replaced by the automatic label `users` and one copy with `LABEL` replaced by the partial label `users.sudo`.

6.4.5.6 Triggering one service restart for multiple files

Each stanza in rdist can push a file or files to a known location and you can run a command if the file(s) is updated. But what do you do for services like httpd/Apache that have multiple control files in different locations and has to be restarted if any of them change?
One way of doing it is:

```
httpd:
$C/httpd/httpd.conf -> ${APACHE_HOSTS}
    SAVEINSTALL(/etc/httpd/conf/httpd.conf, 3);
    cmdspecial "/etc/init.d/httpd restart";

httpd:
$C/httpd/conf.d/*.conf -> ${APACHE_HOSTS}
    SAVEINSTALL(/etc/httpd/conf.d/., 3);
    cmdspecial "/etc/init.d/httpd restart";
```

any changes pushed by either stanza causes httpd to restart. This works but is not optimal. What we really would like is to have httpd restart just once if either or both stanza's pushed files. DACS does this using a couple of macros along with the POSTINSTALL-ACTIONS label.

```
httpd:
$C/httpd/httpd.conf -> ${APACHE_HOSTS}
    SAVEINSTALL(/etc/httpd/conf/httpd.conf, 3);
    POSTACTION(httpd_restart);
httpd:
$C/httpd/conf.d/*.conf -> ${APACHE_HOSTS}
    SAVEINSTALL(/etc/httpd/conf.d/., 3);
    POSTACTION(httpd_restart);

# and at the end of distfile.base

POSTINSTALL-ACTIONS:
$C/.empty_file -> ${APACHE_HOSTS}
        FORCERUN ;
        cmdspecial "IFPOSTACTION(httpd_restart); /etc/init.d/httpd restart";
```

The POSTACTION macro registers a named postaction (`httpd_restart` in this case) on the client machine if a file update occurred. Then when the POSTINSTALL-ACTIONS stanza runs (the POSTINSTALL-ACTIONS label is automatically appended to the list of labels by the Rdist command) it uses the IFPOSTACTION command inside a cmdspecial to allow execution of the following commands if the host has the `httpd_restart` action registered.
Note that the IFPOSTACTION macro is a little different from the other macros in that is occurs inside of an rdist `cmdspecial` command rather than replacing an rdist command like the SAVEINSTALL or POSTACTION commands do.
This could be implemented using a post label named `httpd` (see above for label types section 6.4.2). POSTINSTALL-ACTIONS however transcend labels as the label you use to make the change is not the same label that implements the action. For example let's extend this scenario to include another label called software that updates the Apache binary file and shared objects. Using POSTINSTALL-ACTIONS you can run `sudo /config/Rdist httpd software` and still only have a single restart of the Apache process regardless of having a changing binary, shared object or configuration file.

6.4.5.7 Diffing remote files without update

To use Rdist as a mechanism to diff files before replacement, use a stanza with the VERIFYFILE macro. E.G.

```
etc-manual:
$C/etc/services/services -> ${ALL_HOSTS}
   VERIFYFILE(/etc/services);
```

Now run Rdist using: `Rdist -M 1 etc-manual`, it will perform a `diff -u` (unified diff output) between the argument to VERIFYFILE (/etc/services) and the file in the repository ($C/etc/services/services). The `-M 1` is used to stop rdist from running multiple hosts in parallel which results in interlaced output from the diff command that is difficult to read. The `diff` occurs on the remote machine, so it won't work if the host doesn't have diff installed.

6.5 Supporting Legacy hosts that have part of their configuration unmanaged

You might be tempted to use a macro in the distfile that is subtracted from the list of hosts that should receive the file. For example:

```
LEGACY_FIREWALLS = (
fred.example.com
bar.example.com
baz.example.com
)
...
firewalls:
$C/firewalls/minimal -> ${LINUX_HOSTS} - ${LEGACY_FIREWALLS}
   SAVEINSTALL....;
```

This way new hosts will automatically receive the service/file when the host is added to the database, but older hosts will be excluded or *opt out* from the firewall.
This works, but is less than optimal for reason we will discuss below. However if you are going to manually maintain class lists like this make sure to use an *opt out* list and do not use an *opt in* list in the distfile. Use of an *opt in* list means that newly deployed systems have to get configured in two places: the database and the macro in distfile.conf and it is very easy to forget to *opt in* a new host. A distfile entry for for an *opt in* list (that you should never use) looks like:

```
LEGACY_FIREWALLS = (
fred.example.com
bar.example.com
baz.example.com
)
...
firewalls:
$C/firewalls/minimal -> ${LEGACY_FIREWALLS}
   SAVEINSTALL....;
```

If you are going to manually maintain class lists, the *opt out* list is the safer way to manage this.

6.5.1 Why you shouldn't do this

Using a manually maintained list of hosts *breaks a basic tenet* of DACS:

- all config info is defined in one place, the database.

You can't generate a report on the database and determine that a particular host is being treated as a legacy host if the info about the "legacyness" of the host is located only in the distfile. However defining a macro in the distfile is a useful short term measure if everybody knows about it, but you should never add more hosts to it. Only subtract hosts making it ultimately go away (or replace it with a uses/service keyword in the database).
A better way to "unmanage" a host is to define a new uses value: `LEGACY_FIREWALL` and apply it to the hosts you would add to the *opt out* list. This way you can discover all the hosts holding onto legacy firewall configurations using: `dbreport -l -s "uses:/LEGACY_FIREWALL/"`. This does change the rdist rule slightly as you have to use the proper rdist class name:

```
firewalls:
$C/firewalls/minimal -> ${LINUX_HOSTS} - ${LEGACY_FIREWALL_SLAVES}
   SAVEINSTALL....;
```

but it is safer and better able to be audited.

6.6 Troubleshooting distribution errors

Occasionally you may end up seeing errors during distribution.

6.6.1 Rdist/distfile errors

If you get errors like (wrapped for display):

```
localhost: LOCAL ERROR: Error in distfile: line 903:
  /config/httpd/conf.d/host1.example.com/: No such file or directory
localhost: LOCAL ERROR: Error in distfile: line 905: Bad distfile
  options "savetargets".
localhost: LOCAL ERROR: Error in distfile: line 911: Bad distfile
  options "savetargets".
localhost: LOCAL ERROR: Error in distfile: line 915:
  /config/httpd/conf.d/host2.example.com/: No such file or directory
localhost: LOCAL ERROR: Error in distfile: line 917: Bad distfile
  options "savetargets".
localhost: LOCAL ERROR: Error in distfile: line 923: Bad distfile
  options "savetargets".
```

generally the first error generated is the real error. The others are bogus errors caused by the first one. Note that the line number presented is (903) is in the generated Distfile, and not in distfile.base. So you need to go to line 903 in Distfile, determine what the rdist stanza is, and fix the corresponding stanza in distfile.base. (As a future enhancement, filepp defines the macro `LINE` that is the current input line number. Using this macro provides a way to tie output line numbers to input line numbers and may make locating the entry in distile.base easier.)
To generate the errors again, you can go to any fully checked out copy of the DACS CCM tree change to `Config/distfile/` and run: `make check 2>&1 |head -25` to get the first few errors. You can run this in any working copy of the full DACS repository (note that this may not work on a partially checked out working tree as some files that the distfile wants to push will be missing and thus cause errors of it's own). See the auto-generated Makefile documentation for the entry on check, or check-config.
See the automatically generated documents to find out about other makefile targets such as `audit` in `config/Config/Makefile` or the various check targets in: `docs/distfile/Makefile.html` for additional features that provide information and verifcation before checking in changes to distfile.base. These are also discussed below in "Distribution Reports" section 6.7.

6.6.1.1 Errors with 'label not defined in the distfile'

Two things to check:

1. the label you specify is in the distfile

- grep for 'Îabel:' in the distfile to make sure there aren't weird non-printable characters in the label.

2. a label is defined only if it's applied to a host, so make sure that at least one host in the stanza is valid. It is not an error for the list of hosts to evaluate to the null set, and that label will be removed from the parsed result.

6.6.1.2 Other 'unexplainable' errors

For general troubleshooting, strip all entries (but not variable definitions) out of the file except the ones causing the trouble. See if that makes the problem go away. If it does, see if this test case is easier to analyze. If it's not showing a problem, try adding in the rules before the troublesome ones. Changing the order of the stanzas may "fix" the error.
I agree this is an unsatisfying and worrisome condition, but on some operating systems there appear to be rdist issues associated with the ordering of stanza's. A distfile that works fine on one OS version, may have an issue on another OS or version. I assume the problem is in the libraries that rdist uses, however I can't rule out some odd bug in rdist that manifests itself only under certain circumstances.

6.7 Distribution Reports

These reports provide information on how the distribution system functions. They should not be confused with reports from the database like wiring or asset reports that are discussed in Standard database reports: wiring, ... section 3.6 in the DACS database chapter.

6.7.1 Host-target report

This report identifies distfile.base rules that use host names rather the rdist classes/variables. This identifies places in the distfile.base that have to be changed when services move because simply making database changes won't change the rule's operation.
The host-targets report is performed by changing the the Config/distfile directory and running `make host-targets`. It reports the rdist stanza's containing host names. As an example:

```
etc:
$C/etc/sysconfig/nfs -> ${LINUX_NFSSERV_HOSTS} - ( store.example.com )
        SAVEINSTALL(/etc/sysconfig/nfs, 2);
        cmdspecial "echo 'WARNING: some portions of the NFS subsystem
        must be stopped/restarted to get the effects of the changes to
        /etc/sysconfig/nfs. You must do this MANUALLY.' RDIST_SPECIAL_FILTER";
```

This stanza is produced because of the presence of store.example.com in the stanza. (Note: the cmdspecial is all on one line in distfile.base. It is split here for readability.)

6.7.2 Files-report report

This report provides a map between files in the DACS configuration tree and the target location. The report consists of lines like:

```
a.example.com: /etc/log.d/conf/logwatch.conf <- /config/etc/logwatch/logwatch.conf/logwatch.conf
a.example.com: /etc/log.d/conf/logfiles/. <- /config/etc/logwatch/conf/logfiles/yum.conf/yum.conf
a.example.com: /etc/man.config <- /config/etc/man.config/man.config.through_centos4
a.example.com: /etc/nscd.conf <- /config/etc/nscd/nscd.conf
a.example.com: /etc/hosts <- /config/etc/hosts/dist/a.example.com
...
```

The format is:
`host: file path on host <- source file in the DACS CCM tree`
which can be used to:

- locate the source of a file if you know the location of the file on the target system
- identify files on the target system that are DACS managed (to exclude from a tripwire or AIDE scan for example)
- identify where the source files are distributed to (to verify distfile rules for example)

Files-report is performed by changing to the `Config/distfile` directory and running `make files-report`.
When pushing files to a directory, the target on the left of the `<-` is the directory and not the actual file name. So searching for `/etc/log.d/conf/logfiles/yum.conf` won't work, but grepping for the regular expression: `/etc/log.d/conf/logfiles.*<-.*yum.conf$` should match. (Patches to fix this are welcome.)

6.7.3 Files audit report

This report lets you identify files in the DACS CCM tree that are not used or that are missing. This report is of more use when file generation isn't heavily used and most of the files are manually maintained. But it has a limited value when used with file generation if a convention is established for naming the generated files (e.g. in a dist/ subdirectory).
It generates a three column list comparing the two file lists: one from the distfile and one from the svn command. If a file exists only in the distfile, it is in the far left column and is prefixed with a "d". If it exists only in the source code repository, it starts in the middle column and is prefixed with an "s" for source. If it is matched in both lists, it starts in the third column and is prefixed with a "b" for both. Sample output looks like:

```
s       nsswitch/
b               nsswitch/nsswitch.conf.ldap
b               nsswitch/nsswitch.conf.noldap
s       ntp/
b               ntp/clients.conf
d ntp/dist/ntp_root1.conf
d ntp/dist/ntp_root2.conf
d ntp/dist/ntp_root3.conf
s       ntp/hosts/
b               ntp/hosts/a.example.com.ntp.conf
b               ntp/hosts/b.example.com.ntp.conf
s       ntp/ntp_root.conf
```

Which shows the directory nsswitch present in the VCS list, but not being pushed by the distfile. This is expected and can be filtered by the user. Also the generated files: ntp/dist/ntp_root?.conf are present in the distfile, but not in the VCS. This is also expected as no files under the 'dist' directories should be checked into subversion as they are generated from source files that are version controlled. Files like ntp/clients.conf shows up as being in both VCS and the distfile. These are usually manually maintained files.

Chapter 7

Examples

These examples provide more detail on setting up some simple and more complex configurations in DACS.
They combine and expand on examples from the DacsDatabase, DacsVcs, DacsBuild and DacsDistribution sections. The examples shown here progress from simple to more difficult.
Don't worry if you don't understand some of the more complex examples. Depending on your environment you may not need to use them. However if you work requires high levels of automation and standard policies (e.g. for regulatory compliance) you can automate it using DACS. This section also includes an ambitious use case and some caveats about using extensive automation in "Renumbering a network and a caution about automation".
In the examples dealing with file deployment you will see echoes of the basic 8 steps:

1. Identify what files to manage
2. Decide where to locate them in DACS
3. Decide how to create/manage the content of the files
4. Decide were the files should be distributed
5. Find out if the new files work
6. Commit the changes
7. Test the distribution
8. Deploy to all hosts

In many cases describing the steps is much more time consuming than actually performing them. Initially it will be somewhat time consuming but as you gain experience with DACS, a number of these steps barely take any time at all because you are able to follow prior practice to eliminate some of the steps above.
The detailed version of this list can be daunting, but you have to remember that you are eliminating the need to manually maintain hundreds of files/entries on each host and establishing a documented, reproducible mechanism for maintaining and identifying common elements among these systems.
The detailed list:

1. Identify the files you want to maintain on the client hosts. The biggest bang for the buck are files that are changed on almost every host, especially if changes to those files are frequent, or if bad changes to those files has caused downtime, loss of revenue or unneeded expense. These may be ntp configurations, hosts files, resolv.conf, public key certificate files, ssh configurations that are common across a bunch of hosts. They may also be files that are unique (currently) but that need to be valid in order to properly supply services. E.G. Apache configuration files for the main web site.

2. Figure out where these files will live in the DACS CCM tree. Fortunately with subversion this is easier to change should the initial decision be incorrect, but it is worth a little time to consider: where would other people using DACS expect it to be located. What makes sense for a DACS user.

   (a) Optionally you may want to check the vendor supplied original copy of the file into a vendor branch to have a reference when a newer copy of the file is released by the vendor.

3. How are the files to be maintained, and how many files will there be? Often you don't have just one configuration of a file, you may have two, three or more than one hundred. You can maintain these copies manually, or generate them from a master copy. Once you have more than 5 or 10 configurations, it usually makes more sense to maintain them automatically by generating them from a template, especially if the file changes often or is highly complex. The generation process can simply the file so that the manually maintained file is simpler and removes options from the admin to maintain standards.

   (a) If you are going to manually maintain the files, create and the check in the master copy/copies of the file(s) It is a good idea at this point to try to reconcile the files so that they are as identical as possible in structure. This reduces the amount of documentation and training needed for people who will change the file and reduces the effort required to troubleshoot issues caused by changes.

   (b) If you are going to generate the files, there are more options available. First set up the directory structure with a `dist` subdirectory for the generated (and distributed) files. Then create the manually maintained input files and the build machinery.

4. Decide where the files should be distributed (what hosts should receive the file and what file location e.g. /etc/hosts, /etc/inet/host) and set up the rules to distribute the files from the CCM tree to the hosts.

   (a) Identify the hosts that need a particular file

      i. does a class with these hosts in it already exist?

      ii. can you use a set operation among existing classes to identify the hosts?

      iii. do you need to define a new service, cluster or uses attribute to define a new class of hosts that fulfills number 1 or 2?

   (b) Set up an automatic labeled stanza that pushes the file(s) to the hosts using class variables.

   (c) Does each host need a unique file?

i. If so wrap the stanza in a filepp foreach clause to generate rules for each host to install that host specific file.

5. Test the new files
   (a) If the files are generated or you need to change permissions, run a make to build files/set permissions and manually inspect them to see if they are correct.
   (b) Use the Makefile target `files-report` in the distfile directory to generate the distribution list. This list shows the map from files in the DACS tree to files on the managed hosts. Verify that the files you added are properly listed.
   (c) Manually copy (generated or manually maintained) files from the DACS tree to client systems and deploy them to verify proper operation.
6. Commit the changes to DACS
   (a) Add all newly created files and directories to subversion `svn add`
   (b) Assign permissions to files using svn propset with the `svn:owner`, `svn:group` and `svn:unix-mode` properties if the defaults of root, root and 755 (for executables) or 644 (for other files) are not suitable. Run make to verify that the proper permissions are set.
   (c) Set up ignore lists for work directories like `.sum` and `dist` using svn propset on the svn:ignore property
   (d) Audit files using `svn status` making sure no files that you need have an unknown status (?), are in a conflicted state (C) etc.
   (e) Check-in the changes with comments using `svn ci [list of files]`.
7. Test the distribution
   (a) Run `Rdist -v [label]` and verify that you see the expected files being pushed.
   (b) Run `Rdist -m host label` and verify that the file is installed, activated, and operates properly on a test host. You may want to run this a few times for different hosts or classes of hosts so you can test multiple configurations. This is an automatic version of the steps you performed in item 5.
8. Run `Rdist label` to push the files to all hosts.

One case below discusses the addition of files to a vendor branch. This is useful, but in practice is a pain without automated tools that help with the mechanics. Those tools aren't available, but if anybody would like some help with developing these tools, I would be happy to assist. Because of the lack of tools, a vendor check-in will only be mentioned in one example. See the subversion docs for steps needed to manage a vendor branch if you wish to implement this.
All of the examples below assume you have a working copy of the CCM repository tree already checked out. These examples deal with the decisions needed to implement the scenarios assuming an already deployed DACS system.
It also assumes that you are using a single production configuration tree located under /config and that updates made to the CCM are immediately available there (i.e. there is no promotion or other QA process).

7.1 Changing a managed file

This example is manipulating an already managed file, so a number of the 8 basic steps aren't needed. The ones that are needed are *'ed below:

1. Identify what files to manage
2. Decide where to locate them in DACS
3. * Decide how to create/manage the content of the files
4. Decide were the files should be distributed
5. * Find out if the new files work
6. * Commit the changes
7. * Test the distribution
8. * Deploy to all hosts

In your working copy of the DACS tree, change to the directory that contains the file. Step 3 above changes to:

- Perform an `svn up` in the directory to get any new changes that may have been committed by others.
- Change the file.

then test the file by copying it to the working location. Once it is tested, check it in using `svn ci filename`. Run `/config/Rdist label` to push the change where the label is the name of the top level directory in the working copy.
Because we are updating a file that is already under DACS management, we don't have to make any changes to `distfile.base` as the file is already accounted for.

7.2 Adding a new file (simple case)

This assumes a few things:

- you know what file you want to manage (basic step 1)
- already have a suitable directory under the DACS tree for the file (basic step 2)
- a class is already set up with the list of hosts that should receive the file (basic step 4)

in this example we will push `/etc/ssh/sshd_config` to our hosts. We have a properly configured file on the host `s3.example.com` that we want to deploy to all hosts running the Centos operating system. (Also assume that you have only one major release of the Centos operating system deployed (i.e. Centos version 5.x)).

7.2.1 File installation (basic step 3)

The `ssh/sshd_config` directory under the DACS root already exists (assuming you imported the DACS 2.0 release tree). Change to that directory and copy (using scp for example) `s3.example.com:/etc/ssh/sshd_config` to the file `sshd_config.centos`. Add the file using: `svn add sshd_config.centos`.

7.2.2 Distfile.base setup (basic step 4)

Edit Config/distfile/distfile.base and find the other `ssh:` labels. Then add:

```
ssh:
$C/ssh/sshd_config/sshd_config.centos -> ${CENTOS_HOSTS}
   SAVEINSTALL(/etc/ssh/sshd_config, 3);
   cmdspecial "/etc/init.d/sshd restart";
```

7.2.3 The finish (basic step 5, 6, 7, 8)

Step 5 is done already since the file works on `s3.example.com`, it is already tested. Also all the hosts are running the same OS and therefore the same version of ssh so we expect the file will continue to work on the other systems.
Check in the changes using:
`svn ci ../../Config/distfile/distfile.base sshd_config.centos`
Note your paths may be different depending on the current working directory when you run the command. The example above assumes you are still in the sshd/sshd_config directory.
Add your comments for the check-in and exit the editor. Now run:
`sudo -S c /config/Rdist -S c -v ssh` you should see something like:

```
a.example.com b.example.com c.example.com
d.example.com s3.example.com
    /config/sshd/sshd_config/sshd_config.centos: need to update
    cmdspecial "/etc/init.d/sshd restart"
```

where all the Centos hosts are listed (including the machine you took the file from originally (s3.example.com) since the datestamp of the file is different from the one in DACS). Then to deploy run: `sudo -S c /config/Rdist -v ssh` and you will see:

```
a.example.com b.example.com c.example.com
d.example.com s3.example.com
    /config/sshd/sshd_config/sshd_config.centos: updating
    Starting sshd: [  OK  ]
    Stopping sshd: [  OK  ]
    cmdspecial "/etc/init.d/sshd restart"
```

and that's it. You just set up 5 systems with the same sshd configuration file and activated the changes. When you set up a new Centos host it will automatically receive these changes. If somebody modifies the sshd_config file on one of the hosts, Rdist will report that the file needs to be updated and you can:

- Investigate to find out why an unauthorized change occurred or

- Remember that a planned experimental change is still in effect and may result in different sshd operation on that host or
- Revert the change and re-establish standard operation

7.2.4 Some Variations

Suppose we have both Centos 4 and Centos 5 hosts. These two versions run different versions of ssh, so we can't use just one file as the Centos 5 version supports some directives that the Centos 4 version doesn't.
In this case we can create `ssh/sshd_config/sshd_config.centos4` `ssh/sshd_config/sshd_config.centos5` with rdist stanzas:

```
ssh:
$C/ssh/sshd_config/sshd_config.centos4 -> ${CENTOS_4.X_HOSTS}
   SAVEINSTALL(/etc/ssh/sshd_config, 3);
   cmdspecial "/etc/init.d/sshd restart";

ssh:
$C/ssh/sshd_config/sshd_config.centos5 -> ${CENTOS_HOSTS} - ${CENTOS_4.X_HOSTS}
   SAVEINSTALL(/etc/ssh/sshd_config, 3);
   cmdspecial "/etc/init.d/sshd restart";
```

where the Centos 4 copy is pushed just to the Centos 4 hosts and the Centos 5 copy is pushed to all Centos hosts except Centos 4 hosts (the '-' sign specifies a difference operation between the two sets/classes). So a Centos 6 host for example would get the same file as a Centos 5 host. This may or may not be correct as Centos 6 may have an even newer version of sshd that would require yet another config file variant, but that can be detected when Centos version 6 comes out.

7.3 Adding a new host

Create an entry in the database for the host. Make sure you have at minimum the following fields defined:

```
Machine =
cluster =
enet =
enet_if =
ip =
os =
rdist = yes
services =
uses =
```

If this host is similar to other hosts at a site, you may be able to copy and paste an existing entry and edit it a little to get a working entry.
In the example above, the `rdist` keyword is set to `yes`. This is fine if the host is set up and is expected to be under constant maintenance.

But if you are just setting the system up and you don't yet want it under constant maintenance, but you do want to be able to use DACS to set up the host, set the `rdist` keyword to `request` as in `rdist=request` to allow you to use =sudo /config/Rdist -m new_host.example.com= to request the host be updated. Don't forget to change it to `yes` when the setup is done so it will be automatically maintained.
If the new machine has multiple IP addresses, add a child entry section 3.2.2 similar to:

```
machine = newhost-newip.example.com
aliases =
ip =
enet =
enet_if =
rdist = no
base = newhost.example.com
```

for each additional address.
Once you have the DACS entry set up, copy the ssh public key for the DACS master server to root's authorized_keys file on the new system. This will bootstrap DACS access to the new host.
If your new host is running the same software and hardware as other hosts, and is co-located with exiting hosts, chances are you won't have to add any new files or change any classes in `dbreport` as the existing definitions will provide all the functionality you need.

7.3.1 Managing the ssh keys

Now if using the DACS supplied ssh key management:

- deploy the keys to the host and
- update all the hosts so they will recognize the new host.

Make a new public/private RSA key pair for the new host by changing to the `ssh/host_keys` directory and type `make <machine name>` where machine name is the name in the database entry (newhost.example.com). This will create two new files: newhost.example.com and newhost.example.com.pub.
Logged into the new machine as root, copy the newly created public and private keys into place (e.g. /etc/ssh/ssh_host_rsa_key and ssh_host_rsa_key.pub) The `hostname.pub` file should be copied to `ssh_host_rsa_key.pub` and `the file =hostname` should be copied to `ssh_host_rsa_key`. Make sure the `.pub` file is mode 444 and the key file is mode 400 and restart the ssh server to use the new keys.
Then check the keys into the VCS (`svn ci` in the `ssh/host_keys` directory). On the DACS master server run: `/config/Rdist -m master.server.name ssh`. This will update the known_hosts file on the master server so it will recognize the newly added machine.
On the DACS master server run: `/config/Rdist -S c ssh` to update the keys on all the other hosts (including the new host you just added if it is configured to receive updates by default).

7.3.2 Updating the host

Since the DACS master host received the new known_hosts file in the update above you can run `/config/Rdist -m <new machine name> ssh` on the DACS master. It should update the new

host without asking to confirm the host's ssh keys (if this was not done by the enterprise wide ssh push above). This update will push the keys and config files onto the new machine synchronizing the timestamps on the files.
Then run `/config/Rdist -v -m <new machine name>` to see what files would be updated. Make sure they make sense and if so, use `/config/Rdist -m <machine name>` to push the new files. You may have to run it a few times to get the state to converge especially when Rdist has to create multiple directory levels in which to store files.
Once this is done, if the `rdist` option in the database file for the new machine is set to `request` change it to `yes` so that it will be automatially updated.

7.4 Setting up location specific timezone files

Let's assume that you have two data centers: one in Los Angeles and one in Miami. You want the systems in both data centers to use their local time. Also we will assume that we have only one type of system at each site.
You don't expect this to change much so you are going to place the files under the `etc/` directory in your DACS tree.
Create a new directory etc/timezones. Copy the timezone file for the Miami systems into etc/timezone/timezone.EST5EDT. Copy the timezone file for the LA systems to etc/timezone/timezone.PST8PDT.
Add the directory and the two files to subversion using `svn add etc/timezone`. An `svn status etc` should now show:

```
A etc/timezone
A etc/timezone/timezone.EST5EDT
A etc/timezone/timezone.PST8PDT
```

This example assumes that you have already defined two cluster values `site_lax1` and `site_mia1` by editing `Config/bin/dbreport` and modifying the `%cluster_list` associative array. If you don't have these cluster values established, you should do it now and add the corresponding cluster value to each host in your database. If this is already done, you don't need to do any more work to identify the target hosts for each file.
Edit `Config/distfile/distfile.base` and search for the start of the `etc` labels section (I assume all the stanzas for a particular label are grouped together. This isn't required but makes it easier to view the group.) Somewhere in that section (I recommend maintaining alphabetical order in each labeled section by the source file name as much as possible) add the following stanzas:

```
etc:
$C/etc/timezone/timezone.EST5EDT -> ${site_mia1_C_HOSTS}
    SAVEINSTALL(/etc/timezone, 2);

etc:
$C/etc/timezone/timezone.PST5PDT -> ${site_lax1_C_HOSTS}
    SAVEINSTALL(/etc/timezone, 2);
```

commit (check-in) the files in using subversion:

`svn ci Config/distfile/distfile.base etc/timezone`
enter comments to describe why these changes are being done and what request/problem spawned this decision and solution.
Then run `sudo /config/Rdist etc` and you will see the timezone files pushed to the end systems. (You may also see other files under the `etc` label being pushed as well, you can expand this example to use partial labels, or exclude hosts using the `--exclude` option to Rdist.)
One thing to note about the timezone files is that applications that determine their timezone when they start up may still be running with the wrong timezone. So you may need to reboot those systems before the changeover is complete. You could add:

```
cmdspecial "/etc/shutdown -r";
```

to each of the stanza's above. However given the disruption a reboot causes, you may be better off adding (wrapped for display section 6.4.5):

```
cmdspecial "echo 'WARNING: a reboot may be needed to fully \
     implement the timezone change.' RDIST_SPECIAL_FILTER";
```

and a manual process to schedule and perform the reboots as needed.
The down side to this that you won't receive the warnings ever again because the warning command is only run when the /etc/timezone file is updated. Since the file should remain up to date, the warning is suppressed. However using a post command, you can continue to receive warnings until the system is rebooted. So a better way of doing this is using the POSTACTION mechanism and a post rdist stanza. To do this add:

```
POSTACTION(reboot_needed, timezone);
```

to each of the stanzas that update the /etc/timezone file. The POSTACTION macro records the need for a reboot. The 'timezone' option is used to record what the reason is for the reboot. You could also have `POSTACTION(reboot_needed, driver_update)` which records the need for the same action but for a different reason.
Then after the distribution rdist stanzas (which will probably break the alphabetic ordering) add (wrapped for display section 6.4.5):

```
etc:
$C/.empty_file -> ${site_mia1_C_HOSTS} +  ${site_lax1_C_HOSTS}
        FORCERUN ;
        cmdspecial "IFPOSTACTION(reboot_needed, timezone); echo
        'WARNING: pending reboot needed due to timezone change. \
              Run \"Rdist -m this.host reboot-manual\" for this host.'";
```

This will continue to alert until somebody runs the Rdist command to reboot the system. Define the reboot-manual label using (wrapped for display section 6.4.5):

```
reboot-manual:
$C/.empty_file ->  ${site_mia1_C_HOSTS} +  ${site_lax1_C_HOSTS}
        FORCERUN ;
        cmdspecial "IFPOSTACTION(reboot_needed); /etc/shutdown -r \
                +1min \"going down for reboot\"";
```

which will reboot the host if it still needs a reboot. Also make very sure this is a manual label. You really don't want it being an automatic or verify type label.
Note there is no second parameter for the IFPOSTACTION in the reboot-manual stanza. Any reason for a reboot is sufficient to trigger the reboot and without the second parameter, the action is reset to the 'not needed' state.
This could also be done using the POSTINSTALL-ACTIONS mechanism as described in DacsDistribution. POSTINSTALL-ACTIONS is just a different form of post label. However since we only have one file to update, and it is associated with just a single label using a normal post label/stanza will work just fine.

7.5 Setting up a database driven ntp configuration (new top level directory)

In this scenario we will set up an ntp configuration with three top level hosts that get their time from the Internet and all the rest of the systems will get their time from these three hosts. Rather than embedding the ip addresses or host names of the three top level hosts in configuration files, DACS will generate the configuration files from the database.
Setting this up is a fair amount of work, but it means that a junior admin is capable of moving an NTP server from one machine to another by changing two entries in the database. S/he doesn't have to know how to edit ntp.conf files across hundreds of hosts, change firewalls configurations etc. to perform the operation. All those steps are automated within DACS. So the steps to move an NTP server are:

- Edit the database and move the NTPx service from the old host to the new host.
- Run 'Rdist -m newhost ntp firewalls'
- Run 'Rdist ntp'

Also if you have ntp clients that are on external networks (e.g. routers and switches) listing the host in the database and assigning it the 'NTP' uses keyword is sufficient to allow access to the NTP servers. Again the Cisco expert doesn't have to bother with ntp.conf and firewalls. S/he just adds the database entry for their router and says it uses the 'NTP' service.
The environment that this scenario occurs in has two classes of client hosts that should always have NTP access: SOLARIS_HOSTS and LINUX_HOSTS. The clients should receive their time from the three top level ntp servers by default, but we also want the ability to exclude a host in these classes from using NTP services.
The files for this scenario are located in the ntp subdirectory of the distribution tree.
This scenario has two manually maintained files and four generated files. One manually maintained file will be used to generate configuration files for the three top level ntp servers. The other manually managed file will generate the configuration file for all the clients.
Since pushing a new configuration will be made easier by having a single label, create a new ntp directory at the top of the DACS CCM tree to isolate these changes from other directories/labels. As expected the generated files will go into the `ntp/dist` directory.
Before continuing, let's create a list of the files currently being distributed. Change to the `Config/distfile` directory and run `make files-report`. Later we can compare this files-report to the files-report we get after changing the configuration to distribute the new ntp files.

7.5.1 Files and services

In the ntp directory we will create the two managed files:

- ntp_root.conf
- client.conf

and a Makefile that generates the output files. For now I will just use `touch` to create the empty files and will add the contents later. This lets me set up the structure under svn before I get started. Also I can set the ignore flag on the dist directory so that Rdist distributions won't abort due to the generated files in the dist directory.

```
mkdir ntp
mkdir ntp/dist
touch ntp/ntp_root.conf
touch ntp/client.conf
touch ntp/Makefile
svn add ntp
svn propset svn:ignore '*' ntp/dist
```

At this point running an `svn status ntp` should show:

```
A      ntp/ntp_root.conf
A      ntp/client.conf
A      ntp/dist
A      ntp/Makefile
```

7.5.2 Database changes

We need to be able to identify four things:

- three top level NTP server hosts
- the single list of clients that use those servers

Since there is no existing class that identifies the top level NTP hosts, define three new service values: NTP1, NTP2, and NTP3 and add them to `Config/bin/dbreport`. Also create the uses value "NTPCLIENT" indicating that the host should run the NTP service using the three top level hosts. See Deploying multiple redundant servers section 3.5.1 in the DACS Database chapter for the logic behind adding the three NTP1,2,3 services. See Adding/changing cluster, services etc section 3.4.3 for info about adding the NTPCLUSTER uses value.
Once you define the values, in the DACS database assign NTP1 to the services keyword for one host. Assign the NTP2 value to another host and similarly with NTP3. So we now have identified three different hosts that will be the top level NTP servers. A sample NTP server entry may look like:

```
Machine = s2.example.com
ip = 192.168.9.33/24
os = Centos 5.2
rdist = yes
services = SSHD NTP2
```

Because `rdist=yes` and the services keyword includes the NTP2 value, the host will be automatically included in the NTP2_HOSTS rdist class variable. With the settings above, `dbreport -s "rdist:/yes/" db` generates the rdist class variable:

```
NTP2_HOSTS=( s2.example.com )
```

Also assign the NTPCLIENT uses value to all the hosts we want to be time synchronized. A sample NTP client server host entry may look like:

```
Machine = a1.example.com
ip = 192.168.9.12/24
os = Centos 5.0
rdist = yes
services = SSHD
uses = NTPCLIENT
```

Because `rdist=yes` and the uses keyword includes the NTPCLIENT value, the host will be automatically included in the NTPCLIENT_SLAVES rdist class variable. The output of `dbreport -s "rdist:/yes/" db` generates the rdist class variable:

```
NTPCLIENT_SLAVES=( a1.example.com a2.example.com ...)
```

All the class variables are automatically prepended to distfile.base by the procedure used to build the Distfile.

7.5.3 The build system and file generation

This touches on the critical parts of the Makefile. Some details are changed to simplify the example. See the full Makefile in the DACS distribution at `repository/Config/work/ntp/Makefile` for all the details.
(Note there is also a Makefile.cache in the same directory that is instructive if you are trying to understand how the md5sum mediated cache works. Both Makefiles produce identical output, but via different mechanisms. For those interested in the cache mechanism see DacsBuild.)
The first thing to do is define what files will be created by this Makefile. To do this add them as dependencies of the `all` and `verify` targets:

```
all verify: dist/ntp_root1.conf dist/ntp_root2.conf dist/ntp_root3.conf dist/clients.conf
```

so that `make all` or `make verify` will build the 4 distributed files.

7.5.3.1 Gathering data from the database

In the Makefile we need 6 things:

- the host names of each of the 3 NTP top level hosts
- the IP address of each of these 3 hosts (yes we could do without these and just use host names everywhere but that wouldn't be any fun)

The names of the 3 top level hosts can be obtained from the database by listing the hostname when selecting the NTP1, NTP2 and NTP3 services entry. So the following three dbreport queries:

```
HOST1:=$(shell $(DBREPORT) -l -s 'services:/NTP1/|isbase:/yes/' $(DB))
HOST2:=$(shell $(DBREPORT) -l -s 'services:/NTP2/|isbase:/yes/' $(DB))
HOST3:=$(shell $(DBREPORT) -l -s 'services:/NTP3/|isbase:/yes/' $(DB))
```

will return the host name.
These commands make use of the dbreport command to list (`-l`) the name of the host that is running each of the services by selecting (`-s`) the service identifier and returning only the the hostname for the machine that is at the base of the inheritance tree. See the DacsDatabase document for details on using dbreport.
Once we have the hosts defined, we get the internal IP addresses for the machines using the hostnames defined above:

```
NTPIP1:=$(shell $(DBREPORT) -f ip -s 'machine:/$(HOST1)/|ip:/^192.168/' $(DB))
NTPIP2:=$(shell $(DBREPORT) -f ip -s 'machine:/$(HOST2)/|ip:/^192.168/' $(DB))
NTPIP3:=$(shell $(DBREPORT) -f ip -s 'machine:/$(HOST3)/|ip:/^192.168/' $(DB))
```

that selects the internal ip address (ip address starts with 192.168 which is one of the reserved address ranges).
(Note: these values could be extracted in a few different ways. This is just one implementation.)

7.5.3.2 Generating the client config

With these 6 values we can create the client configuration file dist/clients.conf using the Makefile rule:

```
dist/clients.conf: clients.conf Makefile $(DB)
        filepp  -D "Host1=$(HOST1)" -D "Host2=$(HOST2)" \
            -D "Host3=$(HOST3)" -D "NtpIp1=$(NTPIP1)" \
            -D "NtpIp2=$(NTPIP2)" -D "NtpIp3=$(NTPIP3)" \
            clients.conf -o $@.out
        mv $@.out $@
```

where `dist/clients.conf` will be rebuilt if:

- the maintained clients.conf file changes
- the Makefile that builds clients.conf changes
- the database where the NTP server information is stored changes

This rule calls filepp defining the six values Host1 through 3 and NtpIp1 though 3. It puts it's output into the file `dist/clients.conf.out` ($@ is make shorthand for the file you are trying to make). If the filepp command succeeds, the temporary output file is moved into place to create the permanent output file.
The maintained clients.conf file contains:

```
  # Host1 NTP1
server NtpIp1 iburst
  # Host2 NTP2
server NtpIp2 iburst
  # Host3 NTP3
server NtpIp3 iburst
```

when processed by the filepp command line above generates output similar to:

```
  # box1.example.com NTP1
server 192.168.1.1 iburst
  # s2.example.com NTP2
server 192.168.9.33 iburst
  # box10.example.com NTP3
server 192.168.5.1 iburst
```

Note that this example uses IP addresses to eliminate the dependency on internal DNS. So ntp will keep working even if internal DNS fails.
This is just a fragment of the ntp configuration file, but the rest of the file is boilerplate and was obtained from the original `/etc/ntp.conf` provided with the OS.

7.5.3.3 Generating the three server configuration files

We will look at the generation of a file for the NTP1 server since they are all basically identical except for one small change. The Makefile lines that generates a server config file is:

```
FPPDEFS=-D "Host1=$(HOST1)" -D "Host2=$(HOST2)" -D "Host3=$(HOST3)"

dist/ntp_root1.conf:
        filepp -D NTP1 $(FPPDEFS) ntp_root.conf -o $@.out
        mv $@.out $@
```

This passes the three host names onto the filepp command and processes the `ntp_root.conf` file with the macro `NTP1` defined. In the `ntp_root.conf` file we have:

```
#comment make sure host defines are not empty.
#if ! "Host1"
#error Host Host1 not defined or empty
#endif
```

that checks to see if the `Host1` definition is empty. If it is it exits with an error stopping the file generation and halting the file distribution. This is one way of implementing error checking to see if the dbreport commands failed. After this we see:

```
#ifdef NTP1
#
# clock.redhat.org
#
restrict 66.187.233.4 mask 255.255.255.255 nomodify notrap noquery
server  66.187.233.4 iburst
```

```
#endif

#ifdef NTP2
# NL ntp0.nl.net (193.67.79.202)
# Location: NLnet, Amsterdam, The Netherlands
# Synchronization: NTP primary (GPS), Sun/Unix SunOS 4.1.3
# Service Area: The Netherlands/Europe
# Access Policy: open access
# Contact: beheer@nl.net
restrict 193.67.79.202 mask 255.255.255.255 nomodify notrap noquery
server 193.67.79.202 iburst
#endif
```

Since NTP1 is defined on the command line with `'-D NTP1'`, the lines between `#ifdef NTP1` and the matching `#endif` are printed to the output file. Since `NTP2` is not defined nothing is printed for the NTP2 conditional clause. This defines all the time source servers to be used by the NTP1 server. Then we see:

```
#ifndef NTP1
#
# Host1
#
restrict Host1 mask 255.255.255.255 nomodify notrap noquery
peer  Host1 iburst
#endif

#ifndef NTP2
#
# Host2
#
restrict Host2 mask 255.255.255.255 nomodify notrap noquery
peer  Host2 iburst
#endif
```

which sets the three top level servers to connect to each other. Note that `#ifndef` outputs the text only if the value is not defined. So the first conditional block produces nothing, since have a system sync to itself is useless. The second conditional block produces:

```
#
# s2.example.com
#
restrict s2.example.com mask 255.255.255.255 nomodify notrap noquery
peer  s2.example.com iburst
```

as it replaces the `Host2` macro with the name of the NTP2 server establishing a peer relationship for the NTP1 server. Note that we use DNS names here in contract to the client configurations. If internal DNS fails, the internal peering relationships may be interrupted. However the important relationships here are not the internal relationships but the external relationships to higher level servers. For those relationships, using hostnames resolved via DNS is really the only alternative since they are outside of our span of control.
In the DACS distribution, `repository/Config/work/ntp` has a fully working implementation of this system.

7.5.4 Distributing the generated files

Now that the build mechanism is configured to create the files we want to distribute, we need to configure the distribution. Modify the file `Config/distfile/distfile.base` to push the generated files. We don't use host names in `distfile.base` rules, instead we use the 4 rdist class variables:

- NTPCLIENT_SLAVES - for the clients
- NTP1_HOSTS - for the 1 NTP1 server
- NTP2_HOSTS - for the 1 NTP2 server
- NTP3_HOSTS - for the 1 NTP3 server

So the rule stanzas for the clients look like:

```
ntp:
$C/ntp/dist/clients.conf -> ${NTPCLIENT_SLAVES} & ${LINUX_HOSTS}
        SAVEINSTALL(/etc/ntp.conf, 10, compare ) ;
        cmdspecial "/etc/rc.d/init.d/ntpd restart" ;

ntp:
$C/ntp/dist/clients.conf.solaris -> ${NTPCLIENT_SLAVES} & ${SOLARIS_HOSTS}
        SAVEINSTALL(/etc/inet/ntp.conf, 10, compare ) ;
        cmdspecial "svcadm restart ntp:default" ;
```

Where NTPCLIENT_SLAVES includes all the hosts that have the

```
uses= ... NTPCLIENT ...
```

entry in their database definition.
Note that we use the rdist intersection operator `&`. The first stanza applies to NTPCLIENT_SLAVES that are also LINUX_HOSTS while the second stanza applies to NTPCLIENT_SLAVES that are also SOLARIS_HOSTS.
Why is it split into two rules? There are two different source files (the Solaris ntp server doesn't support iburst mode). But even if the client ntp.conf files had identical contents, look at the two SAVEINSTALL statements. The files need to be installed in different locations on the two operating systems. Also the command to activate the changes is different on the two systems.
With the clients taken care of look at the server configurations. In this case all the servers are Linux boxes, so we don't need to intersect anything, but to prevent the mistake that would occur from using a stanza on the wrong host we use the intersection anyway.

```
## top level NTP servers for our systems
ntp:
$C/ntp/dist/ntp_root1.conf -> ${NTP1_HOSTS} & ${LINUX_HOSTS}
        SAVEINSTALL(/etc/ntp.conf, 10, compare ) ;
        cmdspecial "/etc/rc.d/init.d/ntpd restart" ;

ntp:
$C/ntp/dist/ntp_root2.conf -> ${NTP2_HOSTS} & ${LINUX_HOSTS}
```

```
        SAVEINSTALL(/etc/ntp.conf, 10, compare ) ;
        cmdspecial "/etc/rc.d/init.d/ntpd restart" ;

ntp:
$C/ntp/dist/ntp_root3.conf -> ${NTP3_HOSTS} & ${LINUX_HOSTS}
        SAVEINSTALL(/etc/ntp.conf, 10, compare ) ;
        cmdspecial "/etc/rc.d/init.d/ntpd restart" ;
```

and this completes the setup for pushing to the servers.

7.5.5 Testing

Changing back to the ntp directory, we run `make` and look at the 4 generated files to see if they are correct. You could even manually copy them into place on test systems to verify that they are syntactically correct.
To test the distfile.base modifications, change to the Config/distfile directory. Copy the `files-report` file that we created before to `files-report.orig` and run `make files-report` again. Use diff(1) to compare these two files to see what new files are being pushed. If files are distributed that we think shouldn't be distributed, see the DacsBuild section for troubleshooting tips. Once we are happy with the files check-in the changes.

7.5.6 Check in the changes

There are a number of interfaces to subversion. I use the psvn.el interface within Emacs. It lets you mark which files you want to check in which is very useful. In any case, if you are using the command line change to the root of CCM tree and run `svn status`. You should see something like the following:

```
A      ntp/ntp_root.conf
A      ntp/client.conf
A      ntp/dist
A      ntp/Makefile
M      Config/bin/dbreport
M      Config/distfile/distfile.base
```

If you also see:

```
?      ntp/dist/client.conf
?      ntp/dist/ntp_root1.conf
?      ntp/dist/ntp_root2.conf
?      ntp/dist/ntp_root3.conf
```

You need to do an `'svn propset svn:ignore '*' ntp/dist'` to hide those files from the VCS.
Now you can view the differences between the new files and the original versions of the files using:
`svn diff Config/distfile/distfile.base Config/bin/dbreport ntp | less`
If the differences look ok, check-in the changes and added files:
`svn ci  Config/distfile/distfile.base Config/bin/dbreport ntp`
add your check-in comments, exit the editor and this part is complete.

7.5.7 Update the master tree with a new top level directory

In order to support the ability to use DACS on a partially check out master tree (e.g. one that is used for pushing only a subset of files usually with changes delegated to other people) you must manually check out all top level directories. Rdist by itself won't create a new directory at the root of the CCM tree. Since `ntp` is a new directory, we must perform the update by hand before we run the Rdist script. To do this:

```
cd /config
sudo env -i /usr/bin/svn up ntp
```

This uses the env command to unset all the environment variables and run the subversion command to update the ntp directory. This only has to be done once to create the top level directory. After it is created, new subdirectories, changes to files etc. will all be pulled in by a run of Rdist.
When this is done you can use `sudo /config/Rdist -S c -v ntp` to see what files would be updated using the condensed summary format.

7.5.8 Update the hosts

Run `sudo /config/Rdist -S c -v ntp` to see what files would be updated and then `sudo /config/Rdist -S c ntp` to update all the hosts.

7.6 Integrating external tools into DACS

DACS allows file editing to be delegated to a user, but it can also allow a process to access a file. For example at one site all passwords were handled via local `/etc/password` and `/etc/shadow` files. However they also wanted a single password for each account across all the systems. When the user used the password change server to change his or her password, the password change server checked in the new password configuration and invoked DACS to propagate the new password to all the systems.
Another example of an operation you can integrate into DACS is the distribution of a CRL (certificate revocation list) for a Certificate Authority (CA). These lists can be used in many places if you use certificates for controlling access. They are needed by:

- openvpn - to permit only allowed individuals to access the network
- the corporate web server - where people external to the company can verify that s-mime certificates are still valid.
- web servers - when client certificates are used to authenticate users

Since the need for a CRL is bound not to a host but to a service, DACS makes the logical place to perform the distribution as it knows all of the services that are deployed. As new service instances are brought on line, the CRL will be automatically deployed. However you may not want your certifying authority to be under DACS as DACS tree's can be checked out in multiple places and you may not want multiple copies of the certificate authority's (CA) key.

To integrate the CA with DACS, treat the CA as a delegated user with the ability to update the CRL file.
Let's assume that your CA creates the crl in a subdirectory "CRL" of the CA tree. First create an ssh key using `ssh-keygen`. Place the private part of the key in your CA tree. Let's say it is in /usr/lib/CA/id_rsa. Protect the file so that only the user who performs the key signing can access the file (it should have the same very restrictive access permissions as the master key for the CA). On the subversion repository side, set up the ssh key as described under: "Importing the DACS repository tree" in DacsAppendix. But change the tunnel-user to CA. Then set up the authzfile (see SVN Access controls section 4.4.3) with the following:

```
[/Config/work/etc/crl]
CA = rw
```

Now create the directory `etc/crl` to house the list. As the CA user execute:

```
mv CRL CRL.orig
export SVN_SSH="/usr/bin/ssh -i /usr/lib/CA/id_rsa"
svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/work/etc/crl CRL
cp CRL.orig/crl.pem CRL
svn add CRL/crl.pem
svn ci -m "initial check-in of CRL" CRL
```

This will perform a check-in of the crl file. Note the use of SVN_SSH to set the path to the ssh binary and also the location of the key file that should be used for svn operations. Because the check-in message was specified using the `-m` option, no editor is started.
In your CA machinery that creates the CRL, add:

```
export SVN_SSH="/usr/bin/ssh -i /usr/lib/CA/id_rsa"
svn ci -m "CRL update" CRL
```

so that it will check in the crl when it is changed.
Now adding:

```
etc:
$C/etc/crl/crl.pem -> ${VPNOPEN_HOSTS}
   SAVEINSTALL(/etc/openvpn/crl.pem, 3 );

etc:
$C/etc/crl/crl.pem -> ${HTTP_HOSTS}
   SAVEINSTALL(/etc/httpd/crl.pem, 3 );

etc:
$C/etc/crl/crl.pem -> ${WWWMAIN_HOSTS}
   SAVEINSTALL(/var/www/htdocs/crl.pem, 3 );

etc.crl:
$C/etc/crl/crl.pem -> ${VPNOPEN_HOSTS}
   SAVEINSTALL(/etc/openvpn/crl.pem, 3 );

etc.crl:
$C/etc/crl/crl.pem -> ${HTTP_HOSTS}
```

```
   SAVEINSTALL(/etc/httpd/crl.pem, 3 );

etc.crl:
$C/etc/crl/crl.pem -> ${WWWMAIN_HOSTS}
   SAVEINSTALL(/var/www/htdocs/crl.pem, 3 );
```

to distfile.base and checking the file in will distribute the CRL for:

- openvpn security
- httpd security
- publishing to the public

Now this can be enhanced to allow the CA user to log into the DACS master host and run: `sudo /config/Rdist etc.crl` to push the CRL's to all the locations that need it without having to explicitly log into the DACS master host. Enhancing this example is left an an exercise for the reader.
As a hint to the reader, read the manual for sudo paying attention to the ability of sudo to restrict the command and arguments that can be run. Also reading the sshd man page on using forced commands may be of some benefit.

7.7 Adding new file or files (complex case)

This example goes into great detail on deploying a single file `/etc/services` to systems running either Centos Linux or Sun Solaris. It discusses other scenarios to expand the application of the 8 basic steps into more complex scenarios and encodes a discussion on using vendor branches.
As mentioned above the major steps are:

1. Identify what files to manage section 7.7.1
2. Decide where to locate them in DACS section 7.7.2
3. Decide how to create/manage the content of the files section 7.7.3
4. Decide were the files should be distributed section 7.7.5
5. Find out if the new files work section 7.7.6
6. Commit the changes section 7.7.7
7. Test the distribution section 7.7.8
8. Deploy to all hosts section 7.7.8

7.7.1 Identifying the file(s)

Which files on the client system need changing? Do you need to push just a single file, or a group of related files?
Implementing a particular configuration state may result in a number of different files being changed. E.G. setting up secure ldap for password authentication on a host means changing: /etc/ldap.conf, certificate and key files for the ldap server, master keys files and pam/system-auth. All of these have to be pushed to make secure ldap authentication work.
A lot of the work is in identifying the files that need to change. Once that is done we move on to the next step. To help identify changed files, you can set the service up on one host and use find to locate all the files that were modified to implement the change. Then you will automate the distribution of these changes.
In this example we will add just the single /etc/services file to DACS so all the systems can have a single common file. But how may different /etc/services files exist on the client machine? We hope only two (one for Centos and one for Solaris) but is that true? Assuming that that is true we will have to merge the /etc/system files from the two systems to create a common file.
But that assumption may be wrong so we verify it by surveying that file on all the hosts.

7.7.1.1 How many files exist

Often the contents of the file will change from host to host. Some things that can cause the file to be different are:

- different versions of the software (e.g. the /etc/ssh/sshd_config files on Centos 2 systems are different from those on Centos 4 systems because the new ssh version added some keywords to activate new functionality.)
- different role of the host. E.G. the aliases file on a mail delivery host has more aliases and different aliases since it is the end delivery point for email. Alias files on hosts that just forward to the central mail host have a much more stripped down version of the aliases file.
- site specific changes. E.G. if you have multiple sites with local dns resolvers, you may have a site specific /etc/resolv.conf that lists the local site's dns resolvers first. So for each site the ordering (and therefore contents) of resolv.conf are different. But they are the same within a site.
- entropy - especially when hand editing files, spacing, line order, etc can change making auditing of these files difficult. You will often discover misconfigurations in files this way. However you need to try to figure out if the standard file you are putting in will be compatible with the one you are replacing.

To do the survey, you can use dbreport to generate a list of all the hosts under rdist control (-s rdist:/yes/). Then for each host copy the file to the local directory:

```
for i in ‘Config/bin/dbreport -l -s ’rdist:/yes/|isbase:/yes/’ Config/database/db‘
do
scp $i:/path/to/file file.$i
done
```

Then run: =sum file.* — sort — uniq -c -w 5' to produce:

```
 2 17474    338 services.solaris1.example.com
10 34762    334 services.s1.example.com
 1 35185    422 services.s3.example.com
```

the `-w` options tell uniq to compare just the first 5 characters (i.e. the checksum). The output tells you how many files are identical and how many times each file occurs. In the case above we have three unique copies of the services file. Two of them are the same as services.solaris1.example.com and ten are the same as services.s1.example.com. The host s3.example.com is the only one with its particular brand of services file. So it is quite possible that some service on s3.example.com will fail when moved to another host because of a missing entry in the services file.
This is the most time consuming part of the process as you have to make sure that the files you are generating encompass all the functionality that you want. This audit process may result in more files to manage under the CCM and hence more classes of hosts that receive these files. However it is crucial for understanding how complex the environment is and starting to reduce that complexity.

7.7.2 Where should the files live in the DACS CCM tree

The nice part about using subversion is its ability to track directory and other structural changes, so you are able to better organize your files to make things more efficient. The key questions here are:

1. when I configure or change this service in the future do I have one file or multiple files to change?
2. how likely is it that multiple people will be changing the same files (i.e. how static is the configuration).

Some things like password files (in the absence of LDAP or NIS) will be changed often as employees are added/removed or change roles. Other files like `/etc/services` won't be changed often if at all.
So you may want to make `password` a top level directory so that you can push those changes separately from others that don't change. On the other hand you may place the services file under "etc" in the services subdirectory along with other rarely changing files and update the set of files under etc as a single group.
If I am performing an LDAP setup I can:

- put all the files under the top level etc assuming they will be set up once and not changed
- put them in an top level ldap directory so the changes can be pushed independent of other files
- put them in separate top level directory trees:
  - openldap - for the client configuration needed for the software application
  - etc/pam.d - for the client configuration files that allow login and other authentication to occur

- software/schemas - for changes to the openldap database schemas
- servers/ldap - for the files that configure the ldap servers (as opposed to client configurations)

Your layout depends on how you think you workflow will occur. If you have 3 files that need to be updated (e.g. when a new LDAP server comes on line you add the new server to the openldap config files, the pam config files and add a new servers/ldap config file) it may make more sense to have all the ldap files under a single directory (ldap) where they can be generated or searched easily rather than spread in multiple directories in the DACS CCM tree.
With the predecessor to DACS (Config) relocating files in the CCM tree was often an issue as CVS didn't provide good support for moving files and directories while preserving history information. Subversion does better in this respect, so moving files and directories around is easier. However you still have to worry about re-training users after a restructuring so they go to the new location for the files, use new labels to push the files etc.
For our /etc/services file we will place it in the `etc/services` directory under the DACS CCM tree. To set this up:

```
cd etc
mkdir services
svn add services
cd services
```

7.7.3 Automatic or manual file maintenance

The basic ideas behind DACS is that the DACS database should hold all the information about the machines and services at the site. That information should not be replicated in individual configuration files.
Now there is a trade-off. Setting everything up to be database driven requires an investment of time and money in setting up the machinery as well as time to perform the generation of files. Not using the database increase the risk that some file with embedded information won't be changed when it needs to be leading to loss of service and the expense and effort required to recover.
Also there may be more than one way to engineer a solution for the configuration files. Some of these solutions will support future changes to the files more easily, while some will require changes to support future growth and changes.
E.G. the ldap client configuration could extract the names of the ldap servers from the database and generate files for the clients using that information.
However it is also perfectly valid to assign static names to the ldap servers (e.g. ldap1.example.com) and hard-code ldap1.example.com in the configuration files. Then you generate your hosts files (or dns records) from the database to map ldap.example.com to an underlying machine.
If you plan on using ssl encrypted ldap, the method using static names is better as the static names are hard-coded in the ssl certificates. So moving a secure ldap server from one host to another means you don't need to generate new certificates for the new host.
Both implementations accomplish the end result where changes to the location of the LDAP server (move to a new machine) or changes to the IP address of the LDAP server are propagated to the clients without manual intervention, but the first one requires some additional work if you are deploying encrypted ldap.

You can also choose to hard code all of this information in files (ldap.conf or dns) and not use the database at all, but this increases the risk that you will make a change that is incomplete. So rather than having two redundant ldap servers accessible to all clients, only one LDAP server is usable by the clients. This works fine (and is pretty undetectable) until the one usable ldap server crashes, or is taken down for maintenance, leaving you locked out of the systems on your network. Some files don't use host specific information directly, but instead have slight variations based on the software installed on a host. E.G. between Openssh version 3.6 and 3.9 some new directives were added. So while 90% of the file is identical the other 10% means that you have to maintain two files. Or do you? Using filepp and the build system you could use a single master configuration file and generate the two variants. This makes sense if you need to make changes in the 90% of the file that the two hosts have in common. Now for a file that has only two variants and is not expected to change much, setting up a template driven mechanism may be overkill.If you have 10 or 20 variants the effort to change these files correctly, and keep them in sync so that they actually implement the same function is significant. In this case deploying the templating system saves a lot of effort.
There are some files that don't include host specific information and are suitable for hand maintenance. /etc/services is an example of this type of file, so we can manually maintain the file adding new service/port mappings as needed.
Moving forward with the example, we create the file by copying the vendor supplied services files (one from Solaris and one from Centos) to "services.solaris" and "services.centos" respectively and merge them into services.merged. If you aren't using the vendor tree to keep the original copies, execute:

```
mv services.merged services
svn add services
svn ci services
```

to check-in the new services file. If you are going to use the vendor tree read the next section with all three files (services.solaris, services.centos, services.merged) in the current services directory.

7.7.4 Add the unmodified file to the vendor tree

Since we have an unmodified configuration file(s) as supplied by the OS vendor, add them to the vendor tree under the OS in the same sub-location as you are adding it in the rdist tree.
We want to check in a copy of the services file for each release. We already have "services.solaris","services.centos", and "services.merged" in the services directory.

```
svn switch svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3/etc/services
```

The reason to do this is to allow faster updates to a new version of the operating system. To use this:

- install a new operating system
- check out a copy of the vendor tree
- for every file in the vendor tree copy the corresponding file from the new system on top of the orginal copy

- perform an `svn diff` to see what changes have occurred.
- see if any of the changes should be applied to the CCM working tree.

This makes updating files for each new OS release very easy as you have an original copy of the file to refer back to and can apply the differences between the earlier vendor versions of the file and the new vendor version. This example will discuss performing the operation for the /etc/services file.

7.7.4.1 Make the vendor directory

Create the vendor branch for the Centos and Solaris releases replicating the working tree structure:

```
 svn mkdir -m "checkin of services file from centos 4.3 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3
 svn mkdir -m "checkin of services file from centos 4.3 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3/etc
 svn mkdir -m "checkin of services file from centos 4.3 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3/etc/services

 svn mkdir -m "checkin of services file from solaris 10.2 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/solaris_10.2
 svn mkdir -m "checkin of services file from solaris 10.2 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/solaris_10.2/etc
 svn mkdir -m "checkin of services file from solaris 10.2 box" \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/solaris_10.2/etc/services
{\small \begin{verbatim}

(Note newer versions of =svn mkdir= have the =--parents= option that
will create all needed intermediate directories like =mkdir -p=
does. But for earlier versions you need to run all the commands to
create the tree hierarchy.)

---++++ Import the unmodified vendor file(s)

Now we switch our working directory to the newly created service
directories. Because none of the "services.*" files has been checked
in they will still be present after the switch. So run the commands:
{\small \begin{verbatim}
  svn switch svn+ssh://dacsuser@dacsmaster:\
       /repo/Config/vendor/centos_4.3/etc/services

  cp services.centos services

  svn add services

  svn ci -m "original /etc/services file for centos 4.3"
```

then the same is done for Solaris. Note that this will delete the services file we just added. This is expected as the Solaris branch doesn't have it's services file yet. The lines you type are preceded with a '>'.

```
>svn switch svn+ssh://dacsuser@dacsmaster:\
     /repo/Config/vendor/solaris_10.2/etc/services
D    services
Updated to revision 53.
> cp services.solaris services
> svn add services
> svn ci -m "original /etc/inet/services file for solaris 10.2"
```

Note that the check-in comments include the original source of the file. This makes finding it easier when new OS deployments are done and you need to locate the next generation of the file on the new OS.

7.7.4.2 Create and edit new file in working directory

We should be in the etc/services working copy of the DACS tree. We will now copy a services files from a vendor branch, modify it and check it in. Since we switched this directory above, first switch back to the work tree.

```
> svn switch \
svn+ssh://dacsuser@dacsmaster:/repo/Config/work/etc/services
D    services
Updated to revision 54.
```

Copy and check in the file from a vendor copy. Which OS release you use doesn't matter. This simply records a pointer to the file location in the vendor tree. Then update the copy with the merged copy. We will also commit the file at this point, and reserve testing for later. As long as we don't check in the `distfile.base` changes, the file will live in the DACS CCM tree, but not be pushed to the clients.

```
> svn cp svn+ssh://dacsuser@dacsmaster/repo/Config\
       /vendor/centos_4.3/etc/services/services .
A         services
> svn ci -m "RT:2345 adding base revision"
Adding         services/services
Committed revision 55.
> cp services.merged services
> svn ci -m "RT:2345 adding merged revision"
Adding         services/services
Committed revision 56.
> svn log -v
r56 | rouilj | 2008-12-28 15:07:38 -0500 (Sun, 28 Dec 2008) | 1 line
Changed paths:
   M /repository/Config/work/etc/services/services

RT:2345 adding merged revision
------------------------------------------------------------------------
r55 | rouilj | 2008-12-28 14:52:12 -0500 (Sun, 28 Dec 2008) | 1 line
Changed paths:
   R /repo/Config/work/etc/services/services (from
        /repo/Config/vendor/centos_4.3/etc/services/services:56)

RT:2345 adding base revision
```

using `svn cp` first rather than just adding a services file allows you to locate the location of the file in the vendor tree if the working file changes location in the future. Look at the second log message in the `svn log` output, it reports the vendor file where the services file was copied from. Lastly, insure the files have the proper modes. See Setting modes, owner and groups section 5.3 if the standard owner/group of root/root and perms of 644 (unless file is executable in which case the default perms are 755). In this case the default owner, group and mode is fine.

7.7.4.3 Future maintenance using vendor trees

The vendor tree can be used to ease file updates when new OS releases are deployed. When Centos 4.7 is deployed, create the vendor tree for centos 4.7 using the 4.3 tree as the base:

```
svn co svn+ssh://dacsuser@dacsmaster/repo/Config/vendor/centos_4.3
```

copy in all the Centos 4.7 files overlaying the centos 4.3 files. Now rather than checking in the 4.3 tree, we copy the modified tree to the vendor area for centos 4.7 using:

```
svn cp /path/to/working svn+ssh://dacsuser@dacsmaster/repo/Config/\
     vendor/centos_4.7
```

Now we have a centos_4.7 vendor tree that we can use to generate the list of changes between the 4.3 and 4.7 releases. To merge these changes into the working services entry, change to the working CCM tree and run:

```
  > svn merge svn+ssh://dacsuser@dacsmaster:/repo/Config/\
        vendor/centos_4.3/etc/services/services \
        svn+ssh://dacsuser@dacsmaster:/repo/Config/\
        vendor/centos_4.7/etc/services/services
  --- Merging differences between repository URLs into 'services':
  U    services
```

Use `svn diff` to see what changes were merged and edit the changes if needed to make sure they make sense and are valid. Then you use `svn ci` to check in the changes and push the updated file using `Rdist`.
Alternatively you can use:

```
svn diff svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3/\
      etc/services/services \
    svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.7/\
      etc/services/services | \
 patch
```

to add the differences between the two files to the services file. This usually operates exactly the same as `svn merge`, but with subversion 1.5 and newer, you loose merge tracking information. Now supposed the `etc/services` file was renamed at one point to `etc/services.centos`. The `svn log -v` output will still point to the `centos_4.3/etc/services/services` file. But now we want to merge changes in that file between 4.3 and 4.7 into the `services.centos` file. To do this use an explicit target file =services.centos=:

```
> svn merge svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.3/\
        etc/services/services \
      svn+ssh://dacsuser@dacsmaster:/repo/Config/vendor/centos_4.7/\
        etc/services/services services.centos
--- Merging differences between repository URLs into 'services.centos':
U    services.centos
```

Using the patch method instead of a subversion merge, you can specify the name of the file to patch as well.
Reading the subversion book for your release of subversion to get addition tips and tricks to make this easier is strongly encouraged.
This method removes the guesswork when upgrading from one release to the other, but it does impose the burden of maintaining the vendor tree on each OS release (or sub-release).

7.7.5 Distribute from the CCM tree to the hosts

There are two steps here:
1 identify the set of hosts that should receive the files based on DACS classes 2 generate distfile stanzas to install and activate the files

7.7.5.1 Identification of hosts

From the survey of the file on all the hosts done in the file creation section, you should be able to determine how many unique files needs to be distributed.
Now we have to identify the hosts that should get each file. While hostnames can be used directly in Config/distfile/distfile.base as in the following stanza:

```
$C/cron/root/sshdexternal.root -> ( a.example.com c.example.com)
                  install -ocompare /var/spool/cron/root ;
                  special "EDITOR=/bin/touch crontab -u root -e" ;
```

this is incorrect because the host configuration information isn't derived from information in the database.
Thie cron file implements a single function because these two servers have a public ssh server role. If we want to move this role to another machine, duplicate the role to multiple machines, decommission a.example.com, or figure out what machines have this role it is easier to do by looking in the database. That said, sometimes the quick and dirty way suffices. But you shouldn't do it this way as it will cause more work in thelong run by not enabiling you to use the database to solve problems.
What you should do instead is look for existing classes that can be combined using set operators like intersection, difference and union. These techniques are supported by both rdist and filepp (although by different mechanisms). If you can't find any existing classes you can add new classes. See the DacsDatabase documentation and the ntp example above for more details. For this case the SSHDEXT_HOSTS class would seem ideal.
For our /etc/services example, all the servers we have have an /etc/services file in the same format, but the file is installed in different locations so we will create two rules using the Solaris and Centos host classes.

7.7.5.1.1 Create the distfile rule using the classes Now create new rules using the classes you defined or discovered. Edit Config/distfile/distfile.base. Add the label named after the *top level* directory the file is stored in. E.G. If the file is located in: etc/services, use the target `etc:`. See Distfile Label Types and Special Purpose Labels section 6.4.2 for more information. Set the path to the file using a relative path from the top of the checked out tree. Using '$C/' before the path will anchor it to the root of the directory tree.
So for the `etc/services` source file the rules:

```
etc:
$C/etc/services/services -> ${CENTOS_HOSTS}
  SAVEINSTALL(/etc/services, 3);

$C/etc/services/services -> ${SOLARIS_HOSTS}
  SAVEINSTALL(/etc/inet/services, 3);
```

will install the file etc/services/services as /etc/services on the Centos hosts and /etc/inet/services on the Solaris hosts. These two rules distribute the files to the client classes we originally identified.
When you are done, use `make check` in the distfile directory to verify your changes are syntactically correct. You can search the output of `make files-report` see where the source files will be distributed.

7.7.6 Testing the files

In this case, there isn't much to testing. You can hand copy the new services file to /etc/services on a Centos box and to /etc/inet/services on a Solaris box (the two classes of systems we identified originally). Then run some commands 'telnet host smtp' or use a Perl script that uses the services file to resolve a service name.
Once we have tested all the host classes, we are ready to check in the changes.

7.7.7 Commiting the changes

With the tests done commit the distfile.base changes (as the services file changes were already commited).
`svn ci Config/distfile/distfile.base`
and add your comments.

7.7.7.1 Dealing with commit failures

Sometimes the file check-in fails with the error:

```
Sending        Config/distfile/distfile.base
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/Config/work/Config/distfile/distfile.base' in transaction '10018-1'
```

This just means that somebody changed `distfile.base` since you last updated. If you perform an `svn up Config/distfile/distfile.base`, you should see:

```
G    Config/distfile/distfile.base
Updated to revision 10018.
```

which indicates that the change in the repository was successfully merged into distfile.base. At this point you can test the distfile by running another `make files-report` or `make check`. Using `svn diff` to verify that their changes don't overlap with your change is also useful (although svn will usually report a conflict in this case). `svn log -v distfile.base` is also useful to see what the reason was for the other update(s). You can also use `svn diff` to see all the changes from the prior revisions listed in the log compared to your current revision. If the changes merged in look compatible, check in the merged file.
Sometimes you end up getting a conflict where the other users changes overlap your changes. In this case I refer you to the subversion documentation on Basic Work Cycle (`http://svnbook.red-bean.com/en/1.1/ch03s05.html`) so you can resolve the conflict and check in the new file.

7.7.8 Pushing the files

On an DACS master server use `/config/Rdist -S c -v etc` to update the CCM tree and run the update against the hosts without actually pushing the changes to the hosts. You should see a bunch of updates for /config/etc/services/services being pushed out.
Since we believe the services file is correct, use `/config/Rdist -S c etc` to push the files.

7.8 Configuring Solaris Zones or VMware guest OS

In DacsDatabase the use of the `base` keyword to tie together multiple machine entries representing different interfaces/names for a host is described.
One might try to use that for virtual machines or Solaris Zones as well. However Solaris Zones or VMware servers aren't tied to the base hardware in the same manner. I.E. the global zone/host system doesn't know anything about the child zone or guest server's ip addresses. A guest OS in VMware doesn't even have the have the same hardware configuration as that is abstracted by the virtual machine.
To represent these types of dependencies, don't use the `base` inheritance mechanism and instead use either:

- a cluster value: `global_zone_s1_example_com` or `vmware_s1_example_com` for all the child zones or guest hosts that run on the host s1.example.com.
- or linked service/uses keys

To use linked service/uses keys for a VMware host os create a `VMWARE_1` service to identify the first VMware host instance. Then assign that service to the host running the VMware host os. Then create a `VMWARE_1` uses value and assign the `VMWARE_1` uses value to each guest os/server. This method should also work for Solaris Zones as well. To answer the question: If I take the master host offline what hosts/zones do I impact, you can simply query he database for the hosts that use `VMWARE_1`.

7.9 Configuring MASTER/STANDBY services

In many cases a service can run on multiple systems and handle a local site or some particular group of clients (e.g. a local DNS server). However there are also services that run in a cold backup mode and are configured to be started only when the master isn't going to run. This is an example setup of such a master/standby configuration.
(The standby service can also be called a slave service. However DACS uses the term slave to describe any distfile class derived from a uses key. So to prevent confusion and make it a little more explicit that the second service isn't running, the term standby was chosen.)

7.9.1 Conditions

The host box01 run a service that copies files from a remote location to a number of hosts. Initially this was set up with the DACS service name COPYME. Now we want to deploy a backup service in case box01 dies. To prevent duplication of work by pulling the files over a slow WAN connection, box03 is set up with the same service, but the service is disabled. We want to keep all the config files for the service up to date on box03 to allow fast activation of the service, but it should have the service disabled.

7.9.2 Implementation: single master, disabled standby

The service name in DACS was COPYME when it existed only on box01. We will reuse that name since installation of files etc, is associated with the class COPYME_HOSTS. However we add a modifier to distinguish between the master and standby instances.
In dbreport replace the original COPYME service value with two values:

- COPYME_MASTER
- COPYME_STANDBY

Replace the COPYME service tag with COPYME_MASTER in the services entry for box01 in the DACS database. Add COPYME_STANDBY to the services entry for box03 in the DACS database. Any files that have to be pushed to both the hosts should be distributed to the COPYME_HOSTS class in distfile.base. This will most likely be the case from the initial setup.
To prevent the services from activating/starting, you change the services/DACS/template.fpp to read:

```
#if "SERVICES" =~ /\bCOPYME_MASTER\b/
copyme                  service
#endif
```

to enable it only on the MASTER server. If the server is configured to restart the service on reboot, DACS will catch it and complain.

7.9.3 Enabled, but idled standby

Some services require some processing before it is able to take over. If the service is designed to take input, build up state and not emit anything until it is signaled, or an external file is changed to a given value, we have another way of activating the service.
On both the master and standby the service is running so in DACS/services/template.fpp add:

```
#if "SERVICES" =~ /\bCOPYME_/
copyme                    service
#endif
```

to enable it on both the MASTER and STANDBY.
Then you want to either:

- push a different file based on whether the server is in MASTER or STANDBY mode.
- create/delete a file based on whether the server is in MASTER or STANDBY mode.

If you want to push different files use:

```
  $C/.../masterfile -> ${COPYME_MASTER_HOSTS}
         SAVEINSTALL(/file/location,...);

 $C/.../standbyfile -> ${COPYME_STANDBY_HOSTS}
        SAVEINSTALL(/file/location,...);
```

in distfile.base.
If you want to create a file on the master and delete a file on the standby use:

```
  masterfile -> ${COPYME_MASTER_HOSTS}
         SAVEINSTALL(/file/location,...);

 standbyfile -> ${COPYME_STANDBY_HOSTS}
        FORCERUN;
        cmdspecial "rm -f /file/location";
```

One variant on the create/delete file mechanism is to delete the file from every host, not just from STANDBY hosts:

```
 standbyfile -> ${ALL_HOSTS}
        FORCERUN;
        cmdspecial "rm -f /file/location";
```

this incurs a penalty because it runs a command on every host, but it also make sure that a host that has had the COPYME_MASTER service removed from it (rather than changed to COPYME_STANDBY) is put into standby mode.

7.9.4 Multi-master variant

You may also have multiple masters for a service. One running per cluster of systems. E.G. you may have two offices, Miami and LA and each needs a master and a standby service.
To handle this use service keys:

- COPYME_MASTER1 (for the Miami master)
- COPYME_MASTER2 (for the LA master)
- ... (if you had a third site)

for the standbys, you can either number them

- COPYME_STANDBY1 (for the Miami standbys)
- COPYME_STANDBY2 (for the LA standbys)
- ... (if you had a third site)

or just use a single COPYME_STANDBY key.
If you have to push different files in the two sites, you can push Miami files using the distfile rules

- `miamifile -> ${COPYME_MASTER1_HOSTS} + ${COPYME_STANDBY1_HOSTS}`
  pushes miamifile to both sets of hosts.

or

- `miamifile -> ${COPYME_HOSTS} + ${miami_C_HOSTS}`
  push to any COPYME host but just members of the Miami cluster.

The second one is considered easier to understand since it is more obvious what site is being accessed. You don't have to remember that site 1 is Miami.

7.9.4.1 Advantages

Setting up master/standby servers this way has a few advantages since the information is included in the database.
If you want to see what machines are running MASTER services, run:
`dbreport -l -s 'services:/_MASTER/|isbase:/yes/'`
If you want to include a list of all services on any host running a MASTER service run:
`dbreport -f machine:services -s 'services:/_MASTER/|isbase:/yes/'`
If you want to make sure that only one master service is configured for the single master case:
`dbreport -l -s 'services:/COPYME_MASTER/|isbase:/yes/'` will return only one host. Take that command and run it through `wc -w` to see the number of words (one word per host) and verify that there is only one.
For the multi-master case, you can check each MASTER[N] case individually:
`dbreport -l -s 'services:/COPYME_MASTER1/|isbase:/yes/'`
or you can select each site or cluster
`dbreport -l -s 'services:/COPYME_MASTER1/|cluster:/mia1/|isbase:/yes/'`
and verify that there is one host. Note that this also verifies that the right multi-master tag is assigned to a host, if COPYME_MASTER1 was assigned only to a TOKYO host, the query above would return no hosts indicating a problem.
You can also verify that there is (at least) one standby configured at each site using:
`dbreport -l -s 'services:/COPYME_STANDBY1/|cluster:/mia1/|isbase:/yes/'`
and checking for one or more hostnames.
These tests can be added to the Makefile for the service, or to the Makefile for the distfile to run these sanity checks on distribution of services or any distribution.
However there is a caveat to adding these sanity checks automatically. Suppose you are moving the COPYME_MASTER service from box1 to box4. You may want to have COPYME_MASTER

enabled for both boxes temporarily during the transfer (to make sure you don't accidentally shut down the service on box1 before box4 is ready). If you enforce a single MASTER in the Makefiles, you need to temporally disable the check otherwise DACS will prevent the file distribution because of the error.
Properly ordering the changes:

1. Move the COPYME_MASTER from box1 to box4 in the database
2. Manually disable the service on box1 if desired
3. Rdist to box4 to bring its service on line
4. Rdist to all other systems except box1 to set up any client side changes that need to be done for the MASTER changeover
5. Rdist to box1 to disable and unconfigure its COPY_MASTER service

will permit the safety checks to be left in place and allow an orderly move of the service. However it may be more efficient to structure the move in some other way.

7.10 Configuring dynamically reconfigurable services (firewalls, routing,...)

Dynamic services have an initial configuration file that determines the configuration upon boot (/etc/routes.conf /etc/sysconfig/iptables.conf ...). However command line utilities are provided to change the configuration while running without updating the configuration file. As a result verifying the configuration file doesn't verify the running configuration.
The firewalls mechanism provides a closed loop control mechanism that can detect and report changes to the running firewall service. Some other systems provide an open loop control mechanism which is not as resilient and can result in an incorrect set of firewall rules remaining deployed on the system.
To implement closed loop verification in DACS you need to perform four steps:

1. Create or use an existing file that has a standard format to record the running configuration and that can be used to load the configuration on reboot.
2. Generate in DACS a copy of the file(s) in 1 from your managed configuration files using the build system
3. Run a pre-command from DACS/Rdist that dumps the current state of the service to the file(s) in 1.
4. Set up a DACS rule to compare your generated file (in 2) to the file that was updated in 3. If the compare fails, the file will be overwritten and a command can be executed to implement the new configuration.

To allow the operator to determine what will change if a configuration is pushed, we shall implement a verification rule as well. The details of doing this will be discussed below in the context of manipulating iptables firewall rules running on the Centos Linux distribution. Since

all/most Linux iptables installations include the commands iptables-save and iptables-restore this should be adaptable to most systems.
The DACS distribution includes an example firewalls directory with a representative template file called template.fpp and a Makefile to generate per-host firewall config files.

7.10.1 Implementing the file update rules

This is one of the most complex examples because the firewall configuration consists of a standard DACS component and remotely managed firewall tables that are updated dynamically and these updates must be preserved across firewall updates that are generated from DACS.
All the build machinery for the file updates is also used for the verification rules so the section on verification rule implementation is much shorter.
This build machinery is complex because it creates a file for each individual host that receives a firewall configuration and uses the caching mechanism discussed in DacsBuild to cut down on processing time.
If you don't understand this the first time though, don't worry about it. Chances are you don't need it yet. But by the time you do need it and understand it you will be well on your way to DACS mastery.

7.10.1.1 Identify Configuration File

In the case of iptables on RedHat/Centos systems the initial file is `/etc/sysconfig/iptables`. This is the file we will be updating as part of a configuration installation. It is the output of iptables-save and consists of a series of arguments to be passed to iptables.
We will create a new top level directory `firewalls` in the DACS CCM to hold the firewall rule files. This also means that we will create an automatic label in distfile.base called `firewalls` to push these files.
The goal is to produce a copy of `/etc/sysconfig/iptables` for each host from DACS. This can then be compared to the copy of that file generated from the actual running firewall rules.

7.10.1.2 Generate a copy of the configuration file in DACS

There are 4 files we need:

- Makefile - drives the build mechanism
- template.fpp - processed once for each host to generate the per host firewall rules.
- geniptables-save - a shell script that is used to generate the startup firewall rules file from the running configuration preserving local table modifications.
- local-firewall-restore - a shell script that restores the local modifications to a running firewall.

and two subdirectories

- dist - to store the generated firewall files
- .sum - a work directory for the caching mechanism. See the DacsBuild caching section for more information.

Filepp is used to generate per host firewall rules from the template.fpp file based on what services a given host is supposed to be running. The service list is determined from the DACS database and is cached to allow faster operation.
If you are not familiar with the output from iptables-save, a quick summary can be found at:
`http://www.faqs.org/docs/iptables/iptables-save.html`
A portion of the the template.fpp file:

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

#if "SERVICES" =~ /SSHDEXT/
:BLACKLIST - [0:0]
#endif

:IN - [0:0]

...
## ssh rules for ssh blacklist
#if "SERVICES" =~ /SSHDEXT/
-A INPUT -p tcp -m tcp --dport 22 -m state --state NEW -j BLACKLIST
#endif
```

This file is processed for every host that gets its firewall from DACS. The SERVICES filepp macro is a list of services that run on that host.
The first if clause looks to see if the host is running the SSHDEXT service that permits ssh access from the Internet. If it does, then it defines an iptables table called BLACKLIST.
Then it always defines the default iptables table called IN. Then we see another rule that is output if the host is running SSHDEXT and this rule is added to the default IN table and it activates the BLACKLIST table rules if a new ssh connection comes in. So for a host providing the SSHDBLACK service we will have as output:

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

:BLACKLIST - [0:0]

:IN - [0:0]

-A INPUT -p tcp -m tcp --dport 22 -m state --state NEW -j BLACKLIST
```

So adding new rules to this list is as simple as:

- test the rule on a host
- run iptables-save to output the canonical form for the rule
- paste the canonical form for the rule in the proper place in template.fpp and

- optionally setting up a filepp conditional around the line that controls when the rule is imposed.

The processing that this file undergoes is a bit more extensive and includes replacing cdir (/24) notation (which is easy to type) with the netmask format that iptables-save(*) uses in it's canonical form. It also removes blank lines and does other formatting fixups (like appending a blank space). But by the time the processing is done we have a file that is identical to what iptables-save will generate when it is running the proper ruleset. The details of the processing are documented in the sample Makefile and scripts.
(*) Really what is saved by geniptables-save but ...

Another stanza in the template.fpp file is:

```
#####P* iptables/snmp, iptables/port/161
### POLICY
### Access list for snmp servers
### APPLIES TO
### all hosts
### SYNOPSIS
### Allow access to snmp servers from cacti and nagios servers.
### DESCRIPTION
### Allow access to port 161 from cacti and nagios servers.
#####
#foreach SNMP_IP SNMP_COLLECTOR_IPS
-A IN -s SNMP_IP -p udp -m udp --dport 161 -j ACCEPT
#endforeach
```

Now you may have figured out that SNMP_COLLECTOR_IPS is a macro that is supposed to have the IP addresses of the machines that collect SNMP data (nagios and cacti servers). But where does that come from?
Well at the top of template.fpp is the following filepp statement:

```
#include "filepp.classes"
```

The filepp.classes file consists of definitions of network information from multiple hosts like:

- a list of the IP's of all the DNS servers
- a list of the IP's of all the SNMP monitoring hosts

all of this information is combined with information about the host that we are building the firewall rule set for.
In the Makefile the rule to make filepp.classes is:

```
# create filepp.classes by concatenating the contents of each
# explicitly named cache file, using the cache file name as the macro
# name to be defined.
filepp.classes: $(per_cache_sum_files) $(ALL_MAKEFILES)
        @[ -n "$$MAKE_DEBUG" ] && set -xv; set -e; \
        for i in $(CACHE_FILES); do \
```

```
            echo -n "#define `basename $$i` " >> $@.tmp; \
            tr '\n' ' ' < $$i >> $@.tmp; \
            echo >> $@.tmp; \
          done
        mv $@.tmp $@

# make.sum/filepp.classes build filepp.classes
.sum/filepp.classes: filepp.classes
        @$(UPDATESUM) $@ $?
```

The first rule builds `filepp.classes` by looping over each file (in the variable "CACHE_FILES") using the file name as the macro name. The contents of the file are used as the value of the macro. So the file: `.cache/SNMP_COLLECTOR_IPS` with the contents:

```
s1.example.com
s2.example.com
s4.example.com
```

creates the definition:

```
#define SNMP_COLLECTOR_IPS s1.example.com s2.example.com s4.example.com
```

The first rule also uses some magic that isn't shown here, but that dynamically creates rules of the form:

```
.sum/SNMP_COLLECTOR_IPS: .cache/SNMP_COLLECTOR_IPS
        @$(UPDATESUM) $@ $?
```

that makes `filepp.classes` (indirectly) depend on `.cache/SNMP_COLLECTOR_IPS` using an md5 mediated cache. See the DacsBuild.txt for detailed information on how the md5 mediated cache works.
The second rule in this snippet will trigger an update of `filepp.classes` if `.sum/filepp.classes` needs to be made.
Now how is the file `.cache/SNMP_COLLECTOR_IPS` created? The Makefile rule:

```
.cache/SNMP_COLLECTOR_IPS: $(DB) $(ALL_MAKEFILES)
        $(DBREPORT) -f ip \
           -s 'rdist:/yes/|services:/\bCACTI|NAGIOS[0-9]\b/|isbase:/yes/' \
           $(DB) > $@
```

builds `.cache/SNMP_COLLECTOR_IPS` if the DACS database changes or if any of the Makefiles used to build it change.
So we have a chain going from `filepp.classes` to it's components one of which is .cache/SNMP_COLLECTOR_IPS.
Now what triggers building the filepp.classes file? Well the Makefile rule that generates each per host output file is:

```
dist/%: template.fpp $(ALL_MAKEFILES) .sum/% .sum/filepp.classes
        @echo Building iptables list for $*
        @umask 077; [ -n "$$MAKE_DEBUG" ] && set -xv ; set -e; \
          filepp -w -o $@.tmp -m foreach.pm $(HOST_CACHE)/$* template.fpp && \
          sed -f cleanup.sed  $@.tmp > $@.tmp2
        @mv $@.tmp2 $@
        @rm -f $@.tmp $@.tmp2
```

Whew. Ok, this looks pretty bad, but this isn't as bad as it seems. What you need to see are:

- the output file (dist/% for some hostname replaced with %) depends on .sum/filepp.classes.
- .sum/filepp.classes depends on filepp.classes (from the prior makefile snippet)
- filepp.classes depends on .cache/SNMP_COLLECTOR_IPS (indirectly via .sum/SNMP_COLLECTOR_IPS)
- the file .cache/SNMP_COLLECTOR_IPS depends on the database and makefiles.

So in generating the per host files we traverse the dependency chain and incorporate any database changes into the per host configuration file.
Why do it this way? Well the filepp.classes information is used to generate rules that allow access from hosts running particular services to other hosts. E.G.

- allow snmp data collection
- allow dns packets to pass through

By doing it this way I can deploy a new cacti server, nagios server or dns server and push the firewalls label and all the hosts will automatically allow access for the new server. I don't have to remember to edit the firewall rules as everything is driven off the database.

7.10.1.3 Collect the Currently Running Config Using DACS/rdist

Rdist can run any command as part of a special or cmdspecial directive. However there *must* be a file installed in order to run anything. Using the filepp macros, a file installation can be forced using the `FORCERUN` macro. For example (wrapped for display section 6.4.5):

```
#foreach HOST FILEPP_FEDORA_HOSTS FILEPP_CENTOS_HOSTS
  # update the iptables file from current kernel config.
firewalls:
$C/.empty_file -> ( HOST ) - ${NOFIREWALL_SLAVES}
        FORCERUN;
        cmdspecial "/etc/config/bin/geniptables-save > \
           /etc/sysconfig/iptables RDIST_SPECIAL_FILTER";
```

Where NOFIREWALL_SLAVES is generated from the database (uses `... NOFIREWALL ...`) and contains hosts that either do not have a firewall or use a manually maintained firewall. In either case they should not get this DACS provided firewall.
This is an example of a `pre` label or `pre` stanza. Thus this stanza *must* share the same target name (firewalls) as the stanza in the next section, and it *must* occur before the next stanza in the

distfile. The rdist command runs the stanzas in the same order as they exist in the distfile. $C/.empty_file is just a zero length file. It could be a real file, it would just take more time to transfer a non-zero length file.
The executed command (/etc/config/bin/geniptables-save in this case) is responsible for transforming the command used to dump the running state (iptables-save in this case, but it could be /sbin/ip route if you were managing routing tables via DACS or some other command for other dynamic services) into the form that is generated by DACS.
Note also the filepp foreach loop that starts in this block. For each host running the CENTOS or FEDORA operating system, it outputs one copy of the stanza replacing the HOST macro with each machine name. So the output looks like:

```
firewalls:
$C/.empty_file -> ( a.example.com ) - ${NOFIREWALL_SLAVES}
        FORCERUN;
        cmdspecial "/etc/config/bin/geniptables-save > \
            /etc/sysconfig/iptables RDIST_SPECIAL_FILTER";
```

and so on for each host.

7.10.1.4 Push the DACS configuration file to the remote system/reload service

Once we have updated the file from the running config, we push the DACS generated file to the system if it is different. Note that we use the `compare` option since running the geniptables-save command as part of the pre label will have changed the datestamps on the file. If the compare fails, the file is updated and the cmdspecial commands are run.

```
firewalls:
$C/firewalls/dist/HOST -> ( HOST ) - ${NOFIREWALL_SLAVES}
        SAVEINSTALL(/etc/sysconfig/iptables, 3, compare);
        BACKUP(3);
        cmdspecial "/etc/init.d/iptables restart";
        cmdspecial "/etc/config/bin/local-firewall-restore";
#endforeach
```

Here we see the end of the foreach loop that was started in the prior section.
Sample output for one host would be:

```
firewalls:
$C/firewalls/dist/a.example.com -> ( a.example.com ) - ${NOFIREWALL_SLAVES}
        SAVEINSTALL(/etc/sysconfig/iptables, 3, compare);
        cmdspecial "/etc/init.d/iptables restart";
        cmdspecial "/etc/config/bin/local-firewall-restore";
```

Here we see the firewall configuration file for the host a.example.com ($C/firewalls/dist/a.example.com) pushed to /etc/sysconfig/iptables only if the files are different. Because we have locally maintained parts of the firewall config (e.g. the dynamically added rules in the BLACKLIST that corresponds to ssh login failures), the geniptables-save script splits the running firewall setup into two parts:

- DACS controlled (in /etc/sysconfig/iptables)

- locally controlled (in /etc/sysconfig/iptables.local_rules)

the first cmdspecial command loads the DACS maintained rules. The second cmdspecial merges in the local rules (saved by the run of geniptables-save) for the tables currently defined in the DACS controlled iptables config.

7.10.1.5 Test/Check in the changes

Running make in the firewalls directory should create all the firewall files in the dist directory. You can run geniptables-save on the hosts and diff the output against the generated files to verify that they are correct.
When you are satisfied with the generated firewalls you can run:
`svn ci Config/distfile/distfile.base firewalls`
to check-in the changes. Then you need to prime the new firewalls directory by running:

```
cd /config
sudo env -i /usr/bin/svn up firewalls
```

Then you can run `sudo /config/Rdist -v firewalls` if you want to do some more testing, or just run `sudo /config/Rdist firewalls` to push the new firewalls.

7.10.2 Implement file verification rules

The firewalls-verify rules is implemented similarly to the firewalls rules except the file that is updated is not used by the operating system. This is consistent with the requirement that -verify targets never make operational changes to the system. So it should always be safe to run -verify targets.
Since we want to report differences between the DACS and local config, we need two files:

- the DACS file is installed in: `/tmp/iptables.config` rather than `/etc/sysconfig/iptables`
- then currently running config is dumped to `/tmp/iptables.current`.

Now we have to force the update of `/tmp/iptables.current` and the generation of the diff. We can do this in a single stanza with:

```
firewalls-verify:
$C/firewalls/dist/host.example.com -> ( host.example.com ) - ${MANUAL_FIREWALL}
                  FORCERUN;
                  install -ocompare /tmp/iptables.config;
                  cmdspecial "umask 077; /etc/config/bin/geniptables-save > \
                         /tmp/iptables.current RDIST_SPECIAL_FILTER";
                  cmdspecial "diff -u /tmp/iptables.current \
                       /tmp/iptables.config; \
                       if [ $? -eq 2 ]; then exit 2; \
                       fi RDIST_SPECIAL_FILTER";
```

The `FORCERUN` filepp macro forces the cmdspecials to run regardless of whether `/tmp/iptables.config` is up to date or not. Cmdspecials are always run after all files have been installed, and they are run in order, so we get:

1. push $C/firewalls/dist/host.example.com -> some random file. This is always pushed.
2. push $C/firewalls/dist/host.example.com to `/tmp/iptables.config` if the files are different. This will push the file only if the client file is different.
3. execute cmdspecial to dump the running config to /etc/sysconfig/iptables.current (and filter this command from the report due to the addition of RDIST_SPECIAL_FILTER). This will always occur due to 1.
4. execute diff. If there are differences, diff exits with exit code 1. Suppress the non-zero exit code unless it exits with exit code 2 (something went wrong, missing file, permissions error etc.)

the output is in unified diff format where a leading - sign means the line will be deleted, a leading + sign means the rule will be added and a leading space means the line is unchanged and just provides some context around the changed lines.
Sample output from running `sudo /config/Rdist -S v -m a.example.com firewalls-verify` looks like:

```
a.example.com: --- /tmp/iptables.current 2008-12-04 19:46:59.941823400 -0500
a.example.com: +++ /tmp/iptables.config 2008-12-04 19:44:16.904700900 -0500
a.example.com: @@ -30,11 +28,11 @@
a.example.com: -A IN -m state --state RELATED,ESTABLISHED -j ACCEPT
a.example.com: -A IN -i lo -j ACCEPT
a.example.com: -A IN -p icmp -m icmp --icmp-type any -j ACCEPT
a.example.com:--A IN -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
a.example.com:+-A IN -s 192.168.0.0/255.255.0.0 -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
a.example.com: -A IN -s 192.168.12.13 -p udp -m udp --dport 161 -j ACCEPT
a.example.com: -A IN -s 192.168.9.20 -p udp -m udp --dport 161 -j ACCEPT
```

Which shows that the next update will remove a rule (look for the '–' in the 7th line from the top) that allows ssh access from everywhere and replaces that rule (look for '+-' at line 8) with one that permits ssh access only from 192.168.0.0/16.

7.11 DACS Invocation tips

There are a number of useful functions that you can use to generate lists of hosts by:

- the values to the cluster keyword
- the values to the service keyword
- the values to the uses keyword

The bash shell definitions are:

```
bysite ()
{   # select by site value in the cluster keyword
    SITE=$1;
    shift;
    HOSTS=‘dbreport -l -s "rdist:/yes/|cluster:/${SITE}/" "$@"‘;
    if [ -z "$HOSTS" ]; then
        echo "no hosts";
        return 1;
    else
        echo $HOSTS;
    fi
}
bysrv ()
{   # select by service value in the services keyword
    SERVICE=$1;
    shift;
    HOSTS=‘dbreport -l -s "rdist:/yes/|services:/${SERVICE}/" "$@"‘;
    if [ -z "$HOSTS" ]; then
        echo "no hosts";
        return 1;
    else
        echo $HOSTS;
    fi
}
byuse ()
{   # select by uses value in uses keyword
    USES=$1;
    shift;
    HOSTS=‘dbreport -l -s "rdist:/yes/|uses:/${USES}/" "$@"‘;
    if [ -z "$HOSTS" ]; then
        echo "no hosts";
        return 1;
    else
        echo $HOSTS;
    fi
}
```

These can be used to do some simple host selections. For example if you want to distribute files to just APACHE servers, you can use:

```
/config/Rdist ... -m "‘bysrv APACHE‘" ....
```

which is easier than specifying each host using individual `-m` options. To update just the hosts at the lax1 site:

```
/config/Rdist ... -m "‘bysite site_lax1‘" ....
```

To update hosts that use LDAP:

```
/config/Rdist ... -m "‘byuse LDAP‘" ....
```

You can also use these functions in shell scripts entered at the command line as well. If all your hosts run sshd and you want to run a command on all the hosts you can run:

```
for host in ‘bysrv SSHD‘
do
  ssh $host some command
done
```

7.12 Renumbering a network and a caution about automation

A number of years ago I heard about a modified version of Config (the DACS precursor) that was used to manage the renumbering of an entire network of hosts (approx. 1000). As a result of this discussion the current inheritance (via the base keyword) and the support for Ethernet interface names (enet_if) was added.
They set up Config to manage network interfaces with netmasks, routes and IP addresses. To perform the changeover they renumbered the entire network to it's final state in the database and checked in the changes. Starting from the point furthest away from the Config master they updated the hosts and activated the new address configuration by rebooting the hosts
At this point they were unable to contact those hosts until the entire readdressing was done as the hosts did not have valid routes or other network connectivity back to the Config master host. They continued to push out the configuration changes to other hosts and network devices getting closer to the configuration master until the final set of systems and devices were reconfigured.
At which point all the routes established in the Config system were active and the systems started coming on line. Apparently they lost only a handful (10-20) machines some of which failed on reboot. I claim their success had much more to do with the disciplined preparation and inventory of systems and networks than the Config tool. Indeed I would be scared to try that myself and they may well have succeeded despite Config tool. However Config provided a structure and plan of operation for the conversion which may have resulted in their high degree of preparation.
However their deployment does bring up one particularly important issue. Because all of the information in DACS is contained in the database, you can get a final configuration that won't work until all of the parts are deployed. Hence the order of deployment is critical and must be actively managed. E.G.

```
  [Config Master]  -> [router] -> [client host]
```

what happens if the router gets it's new network configuration before the client host? At that point the client host is unreachable and will never be able to get the proper configuration update from DACS.
There are three choices:

- roll back the configuration in the VCS and put the router back to the way it was before the update to establish connectivity to the client host. Then roll forward the configuration so that the Config master now has the new client configuration to use to update the client host.
- manually reconfigure (which required a trip to a remote site) the client host to re-establish enough connectivity to allow DACS to push a full update.

- manually reconfigure the router (which should still be accessible from the DACS master host I hope) to re-establish connectivity.

From what I understood, the automation they had in place obtained the default route for a host from the IP address assigned to the router on that subnet. So simply changing the IP address on the router interface also updated/added default routes for the clients. So to get the right default route on the client host, you modified (one of) the router's IP address which means that the router configuration was also updated in Config to impose the new address. There was no easy way to create a configuration state where the router had it's original IP address but the client hosts got the new router IP address.
Because of this coupling, the order of updates had to be carefully managed. Fortunately using `-m` and `--exclude` (their idea) along with labels they succeeded.
But what does this mean for the less adventuresome?
Looking at the ntp deployment example earlier in this section, what happens if you move the NTP1 token from s1.example.com to s7.example.com and then update the ntp configuration on a client host (say a1.example.com)?
Well a1.example.com will look at s7.example.com for it's ntp service, but s7 doesn't yet have the proper configuration to provide that service. It will work itself out eventually, and in this case s7 is merely one of three servers so operationally there are still two valid servers. But the problem of creating two dependent configurations from a single change still stands.
We can work around this issue in DACS by deploying to s7.example.com first (then to the NTP2/NTP3 servers) and then deploying to the NTPCLIENT_SLAVES. Which avoids the problem by establishing the NTP services before the client use those services.
However if we manually maintained the ntp client configuration, you could check-in the database change and no client configurations would change. Then once the server was configured you would change the client configuration and let everything update.
Having to check-in the intermediate states of the network to the VCS to perform this staged deployment is one solution to the problem, but also one that prevents a high degree of automation.
Another solution is to have some sort of validation mechanism so that the generated client ntp.conf is recognized as invalid and is not activated until the s7.example.com server is synchronized and responds to ntp requests.
I am not sure which is better:

- the ability to explicitly order configuration changes
- the ability to stage intermediate configurations
- the ability to verify invalid updates and not apply them

or if there is an even better way of handling this issue that:

- preserves the high degree of automation without the planning burden associated with having to find an explicit ordering for the updates
- reduces the risk of having incorrect/incomplete configurations caused by lack of automation and storing duplicate information in multiple locations

- eliminates the effort required to write/test/execute validation mechanisms for each individual configuration file and file option before applying them

(Note: if anybody recognizes this story, or the gentleman (possibly named Peter) who came to a laser show at the Brevard Community College planetarium in Cocoa, Florida in 1999. Please get in touch with me as I would like to be able to attribute this properly and be able to gather further details.)

Chapter 8

Advanced Topics

This includes some advanced topics not covered elsewhere.

8.1 Using multiple DACS master trees

There are four reasons to use multiple master trees:

- Split production/work
- Redundancy
- Load Distribution
- Test trees (per admin trees)

Split production/work trees are used when implementing a process for moving tested changes to a production network.
Redundancy is useful in recovering when the main DACS server goes down, or when you have multiple networks or sites and you want to be able to use DACS even if you lose connectivity between networks.
Load distribution is still a work in progress but is designed to handle the scaling issues of DACS to larger networks.
Test trees are used to limit access of DACS to particular hosts in the network. It can also be used to allow DACS to scale to multiple admins in the cases where there is extensive DACS update traffic.

8.1.1 Split production/work CCM master trees

Some sites don't permit changes to be pushed to production without a QA/change review. DACS can work in this mode, however with subversion it is somewhat problematic. The workflow is described for subversion in Workflow in a split test/production implementation section 4.5.1 in the DACS Vcs chapter.
That section also describes the issues involved in implementing this using CVS.
This senario uses multiple master trees similar to the test trees discussed below. The major difference between the multiple master in this senario and the others in this section is that the

source for the files in the split production/work master trees are selected/updated based on different criteria.
In the subversion case, the work master tree where testing is done and check-ins occur is not the source for the production master tree. Files are promoted from the work master tree to the production master tree using a seperate process that can include QA steps.
In the "Test trees" senario below, all the masters share the common work tree, so changes that are checked in will propigate to all the trees.

8.1.2 Redundancy

What happens if your only DACS master machine dies? Well you can recreate it from backups and possibly from individual admin's check out trees. However it is useful to have an automatic mechanism for maintaining a backup DACS master.
If your DACS master host has the service: `DACS1`, and your backup masters host(s) have the service: `DACS2`. Adding a rule similar to (wrapped for display section 6.4.5):

```
#if "THISHOST" eq "FILEPP_DACS1_HOSTS"
POSTINSTALL-ACTIONS:
$C/.empty_file -> ${DACS1_HOSTS}
        FORCERUN;
        cmdspecial "for host in FILEPP_DACS2_HOSTS; do \
             rsync -a --delete --delete-excluded --exclude .locked/ \
                -v /config/. $host:/config/. 2>&1; \
             ssh $host touch /config/Config/database/db; done";
#endif
```

makes any Rdist of the master sync itself to the client hosts.
It uses rsync rather than a nomal rdist install command because rsync is a faster method of pushing the tree. But if you use the most restrictive ssh setup, then you will have to use the slower method by using rdist directly:

```
#if "THISHOST" eq "FILEPP_DACS1_HOSTS"
POSTINSTALL-ACTIONS:
$C -> ${DACS2_HOSTS}
   install -o numchowner,numchkgroup,remove,quiet /config;
   except_pat ( .locked )
   cmdspecial "touch /config/Config/database/db";
#endif
```

which updates the backup master hosts any time Rdist to the backup hosts occur.
The Makefile supplied in the distribution creates the `THISHOST` macro. The `#if` filepp command makes sure that `THISHOST` is the host that is defined as the DACS1 (i.e. master) host. This makes synchronization occur only from the master to the backups (because this rule will be not be present on any Distfile generated on the bckup servers).
We exclude any directories called `.locked` as they are the locks put in place by the running Rdist and we don't want to replicate the locks to the unused replica filesystems.
The touch of the `db` database makes sure that a full rebuild occurs (including the Distfile). Thus the Distfile on the backup system will be rebuilt if it is used to distribute files.

The install command will make a backup copy of the whole tree to one or more machines. Now certain configuration files on the clients like:

- root's /.ssh/authorized_hosts file
- hosts files

and maybe some others depending on your local setup will have to be modified to allow both the primary and backups DACS masters to ssh in as root on the clients.
Don't make the mistake of thinking that you can use DACS to switch to the backup DACS master server. This works only when you have a controlled changover. If you have switch due to a machine crash you will be out of luck.
The default distribution includes distributing the id_rsa key file from the master DACS host to master and standby DACS replicas so that key changes from the master will propigate to the standby replicas. You may wish to use different keys for each DACS replica.
If you are using the rdist install command to maintain your backup servers, note that it doesn't save backups because the backup files would cause an Rdist to fail. It uses `remove` to delete any files that have been removed from the master tree. It also uses `numchkowner` and `numchkgroup` so that it creats a uid/gid identical copy of the files. If your master has a different uid/gid mapping from the backup you will have issues. However if you maintain constant uid/gid mappings even if the backup server is missing some mappings the sync will still work right. However you should maintain identical configurations of

- password and group files
- installed software (filepp, make ...)

among the backup masters.
You will also want to replicate your svn repository. More info on this is located in DacsAppendix. Also there are some excellent articles on the web about using subversion replication, or you can use rdist after locking the repository from check-in to make sure you get a consistant copy. If you lose your svn repository you will need to perform an `svn switch` operation as root to move to the backup svn repository before you run Rdist from the backup server. this is detailed in the DacsAppendix.
If you are going to maintain backup servers, I strongly urge you to run `Rdist -v` operations on a regular basis from all your backups and compare them to the `Rdist -v` report run on your master. This can identify failing disks (corrupting data), incorrect setup and other problems before you find out your backup DACS server won't work.
Some sites use the VCS mechanism to replicate files rather then pushing a processed tree. In this case the stanza looks like

```
POSTINSTALL-ACTIONS:
$C -> ${DACS2_HOSTS}
   FORCERUN;
   cmdspecial "svn up /config";
```

this does perform the updates successfully (although it may time out if there is a large number of files to update). But because the other config tree is totally seperate, the modification dates on

the files updated by svn are different. So pushing from that tree will result in regenerating and upating more files than the method that copies the master CCM tree.
There is one issue with this setup. If the master or backup hosts are not distributed to, then the updates won't occur. So if you update some set of hosts that doesn't include the DACS master, those updates won't be distributed to the backup master hosts. One way to handle this is to create an alias for the master host and modify Rdist to make sure that the host is always updated. For example add an entry for the machine `masterhost` to the DACS database, and add the stanza:

```
#if "THISHOST" eq "FILEPP_DACS1_HOSTS"
POSTINSTALL-ACTIONS:
$C/.empty_file -> ( masterhost )
        FORCERUN;
        cmdspecial "for host in FILEPP_DACS2_HOSTS; do \
           rsync -a --delete --delete-excluded --exclude .locked/ \
                 -v /config/. $host:/config/. 2>&1; \
           ssh $host touch /config/Config/database/db; done";
#endif
```

to `distfile.base`. Also add `-m masterhost` to the definition of "@DEFARGS" in the Rdist command. This will distribute to `masterhost` anytime Rdist is run.
You can do something similar for the DACS2 hosts if you use the rdist "install" command to sync the changes.

8.1.3 Load distribution (work in progress)

A similar although different reason for having multiple DACS CCM master trees is to distribute the load. Since DACS is a push system it can take a while to update thousands of hosts. If you are running across a WAN, it would be nice to be able to push one copy of the files to each remote site and update the hundreds of hosts at each site from a local master.
I have not used this setup, but from what I understand the basic idea is to set up replication like above, and trigger a subset of hosts to run from each location. Then a series of distfile stanzas like (wrapped for display section 6.4.5):

```
Distribute:
/tmp/rdistargs -> ${DACSDIST_HOSTS}
   install /tmp/rdistargs;

Distribute:
$C -> ${DACSDIST_HOSTS} & ${site_lax1_C_HOSTS}
   install -o numchowner,numchkgroup,quiet /config.lax1/. ;
   cmdspecial "SUDO_USER=${SUDO_USER}; export SUDO_USER; \
         /config.lax1/Rdist `cat /tmp/rdistargs`";

Distribute:
$C -> ${DACSDIST_HOSTS} & ${site_mia1_C_HOSTS}
   install -o numchowner,numchkgroup,quiet /config.mia1/. ;
   cmdspecial "SUDO_USER=${SUDO_USER}; export SUDO_USER; \
         /config.mia1/Rdist `cat /tmp/rdistargs`";
```

```
Distribute:
$C -> ${DACSDIST_HOSTS} & ${DACSMASTER_HOSTS}
   FORCERUN;
   cmdspecial "SUDO_USER=${SUDO_USER}; export SUDO_USER; \
        /config/Rdist `cat /tmp/rdistargs`";
```

where /tmp/rdistargs is a copy of the command line passed to the Rdist command except for the –distribute flag (note the DACS 2.0 Rdist command doesn't have any of this functionality). The SUDO_USERS variable is needed by Rdist to emulate the environment that would normally occur when it is run under sudo. It propigates the username of the user on the master system so it can be used for logging or other purposes.
Then the Rdist command runs 'rdist Distribute' as a result of being passed the –distribute option to start the distribution.
Each tree is pushed to a different root location. Then this root location is used an an index into the `%config_roots` associative array in Rdist. This array sets arguments for dbreport to limit the hosts that are returned. If the host is not reported from dbreport, it can't be updated using Rdist. As an example you would edit Rdist to add the entry to the `%config_roots` array

```
 '^/config.mia1$' => '|cluster:/site_mia1/',
```

to permit only hosts with the site_mia1 cluster value if the root of the config tree is /config.mia1. Thus the miami site's master pushes only to it's own hosts.
To pick up the hosts that are not part of any submaster the final Distribute stanza triggers (and is missing –distribute so it acts just like a normal distribution) you set up an entry:

```
 '^/config$' => '|cluster:/site_nyc1/|cluster:/site_bos1/',
```

that permits the master DACS server to update all the hosts except the ones handled by the distributed hosts. Note that besides geographical distribution you could restrict by organization boundaries using something like 'cluster:/qa/', 'cluster:/dev/' etc. (See Test Trees section 8.1.4 for a more descriptive example of changing the %config_roots associative array.)
Now while this is doable, the current DACS Rdist is not set up to do it in a reasonable fashion. Also the reporting of updates is made more confusing by the multiple layers of distributing hosts. If you are working at a site that has over 300 hosts, you may want to consider an alternate CCM system to handle these scaling issues.

8.1.4 Test trees

The directory that Rdist is in determines the distribution tree it uses as well as the list of hosts that it can distribute to.
This is an easy way of allowing junior admins to test changes on a subset of hosts without being able to push them to all the hosts.
Any directory tree can be set up to limit it's access to a subset of the hosts in the database. When running `sudo /config.test/Rdist...` the only hosts that can be updated are hosts with the TESTHOST service defined in the database.

There is one location for a test tree defined by default. It should be checked out to `/config.test`. (See Creating DACS CCM master trees section 9.4 for how to check out a DACS CCM master tree.)
If you want to check out another tree at say `/var/DACS/test` you can establish this by modifying the %config_roots associative array. Add a line like:

```
'^/var/DACS/test\$'   => '|services:/TESTHOST/',
```

after the default line of:

```
 'the_default'   => '|machine:/NoSuchHost/',
```

and before the close parenthesis ')'.
The line you added consists of a key and value separated by an arrow `=>` . The key is matched against the path used to call the Rdist command and the value is used to modify the `-s` option in a dbreport call. So the `|services:/TESTHOST/` addition will only allow the selection of hosts that have a service TESTHOST.
Note this mechanism *DOES NOT* prevent somebody with access to all hosts from pushing the changes done by the junior administrator as both the restricted and full access trees follow the head version of the same repository.
To prevent a junior admin from checking something in that is not available for production immediately requires a split work/production senario to be set up in the version control system. This setup is described above and in Workflow in a split test/production implementation section 4.5.1.
Since Rdist locks the repository to guard against unsynchronized changes a single CCM master tree only allows serial access. Using the test tree mechanism (with or without the host restrictions), multiple admins can work in parallel since each master tree is independent of the others and the locks done by Rdist are unique to each tree.
In addition this mechanism may be useful when delegating file changes to a third party. If these file changes are destined for a limited number of systems, a restricted master tree can simplify allowing that user the ability to change and distribute those files.

8.2 Handling network devices

Handling network devices looks a lot like handling a dynamic service section 7.10 with one twist as in both cases you need to:

1. Have the device create a file with a standard format that records the running configuration and that can be used to load the configuration on reboot.

2. Generate in DACS a copy of the file(s) in 1 from your managed configuration files and build system

3. Run a pre-command from DACS/Rdist that dumps the current state of the service to the file(s) in 1.

4. Set up a DACS rule to compare your generated file (in 2) to the file that was updated in 3. If the compare fails, the file will be overwritten and a command can be executed to implement the new configuration.

The trick is that network devices aren't general purpose computers. The configuration file generated by step 1 is usually obtained from the console or something insecure like tftp and requires a login to the device or snmp access or some other method of interaction besides ssh. Also almost none of these boxes will run rdist (well some linux based terminal servers have a developer kit which can be used to build an rdist client), so some alterate management mechanism is needed.
Usually it takes the form of subsidiary scripts that are run on a proxy host. The proxy host is a host that is a standard DACS client, but that can access the network device.
Let's look at the setup of just such a proxy using the DACS master host. To do this you create a dummy hostname for the proxy host. E.G. to manage the device ciscortr01.example.com, you use the host name: `=ciscortr01.example.com.proxy=`. Then in your /etc/hosts file use:

```
127.0.0.1 ciscortr01.example.com.proxy
```

to push files destined for the Cisco router to the local host.
In the database you use:

```
Machine = ciscortr01.example.com
ip = 192.168.9.1/24
os = IOS 12.3.6
rdist = yes
services = ROUTER
uses = PROXY
```

this will place the router in the classes:

- ROUTER_HOSTS
- PROXY_SLAVES
- IOS_HOSTS
- IOS_12.X_HOSTS
- IOS_12.X_HOSTS
- IOS_12.3.X_HOSTS
- IOS_12.3.6_HOSTS

Then you can create rdist stanza's like (wrapped for display section 6.4.5):

```
#foreach CISCO FILEPP_PROXY_SLAVES
  #if " FILEPP_IOS_HOSTS " =~ / CISCO / &&  \
      " FILEPP_ROUTER_HOSTS " =~ / CISCO /
  # only applied for PROXY_SLAVES that run IOS and are routers
  #  (i.e. are Cisco routers)
```

```
  # pre command to dump the running config
cisco:
$C/cisco/CISCO -> ( CISCO.proxy )
FORCERUN;
cmdspecial "/etc/config/bin/cisco-access getrunning \
             /dist/cisco/CISCO.config";
  # if the running config is different from the one in DACS push the
  # DACS version and update the cisco.
cisco:
$C/cisco/CISCO -> ( CISCO.proxy )
SAVEINSTALL(/dist/cisco/CISCO.config, 10, compare);
        cmdspecial "/etc/config/bin/cisco-access setrunning \
               /dist/cisco/CISCO.config";
        cmdspecial "/etc/config/bin/cisco-access setstartup \
               /dist/cisco/CISCO.config";

  # pre command to dump the running config to some alternate location
cisco-verify:
$C/cisco/CISCO -> ( CISCO.proxy )
FORCERUN;
cmdspecial "/etc/config/bin/cisco-access getrunning \
               /dist/cisco/CISCO.config.verify";

  # diff the two Cisco configs if they are different. The version
  # dumped  from the router is in the SAVED file.
cisco-verify:
$C/cisco/CISCO -> ( CISCO.proxy )
SAVEINSTALL(/dist/cisco/CISCO.config, 10, compare);
        cmdspecial "diff -u
    /dist/cisco/CISCO.config.verify.SAVED \
            /dist/cisco/CISCO.config.verify";
  #endif
#endforeach
```

So we use a pre command to dump the running configuration on the device, if that diferes from the version in DACS the file is updated and we execute commands to either diff the files (in verify mode) or put the configuration (in update mode)
The script that talks to the device (cisco-access in this case) can be written in expect, perl or any other language and can use ssh, telnet or some other protocol to perform the update. Also it can use information from the database that is generated and pushed using a pre command.
The access command needs the following functions:

get gets the configuration in standard form and saves to a file

set takes the configuration in standard form and loads to the

activate a command to activate what was installed by the set command if set doesn't activate it immediately.

verify takes some external information about the device (IOS version, hostname etc) generated from the database and verifies that it is valid.

since some devices differentiate between a running and startup configurations and they can be set seperately it may be useful to add those as special cases. Also the access command is repsonsible for any manipulations needed to get the dumped configuration into a standard form (e.g. remove any datestamps in the output, clear counters if they are included ....). It may also be responsible for merging the DACS supplied config and the current running config (which is available in the `.SAVED` backup file). An explict difference or change mode may be needed if the device can't dump a textual version of the file.
Using a tool like NS4 to build the access methods may be a useful shortcut. Cisco's lend themselves to this method of management quite well as they have a text based config that is easily parsed.
The DACS version of Rdist doesn't support proxy operation just yet. There are two changes that are needed to make this work cleanly:

1. in the database add the keyword proxyhost which will hold the proxy hostname (and it can validate the proxy hostname againt a valid host).

2. change Rdist to substitue for the real hostname the proxyhost name, so it would change `Rist -m ciscortr01.example.com` to `rdist -m ciscortr01.example.com.proxy` when it calls rdist(1) internally.

These changes haven't been done yet in any released Rdist version.
Supporting the automatic generation of the hosts file to map proxy hosts to IP addresses can be done using the standard DACS mechanisms:

```
Machine = ciscortr01.example.com.proxy
ip = 127.0.0.1/32
os = NS 4.2
rdist = no
services = PROXY
```

where NS 4.2 indicates that it runs ns version 4.2 and may be useful for further validation. However generating the hosts file is now as simple as: `dbreport -h -s 'services:/PROXY/'`. Note that `rdist=no` in the database stanza for the proxy. This is fine as the original `ciscortr01.example.com` is set up to use rdist, and Rdist (when it has proxy support will) transparently replace the original host name with the proxy name.

Chapter 9

Appendices

The appendicies include operational aspects of DACS that are handled outside of day to day DACS operation.This includes:

- inital setup and importation of the DACS distribution
- setting up ssh access between hosts
- creating working trees and CCM master trees
- setting up replication
- links to other documentation

9.1 Importing the DACS repository tree

The repository subdirectory of the DACS distribution is designed to become the root of a repository and it includes support for managing:

- services under linux and solaris
- firewalls under linux
- ntp files
- root id_rsa and id_rsa.pub keys
- some additional files

as well as a starter set of cluster definitions, a sample database and subversion hooks scripts and authentication files.
Note that this tree includes id_rsa and authorized_keys files in the repository/Config/work/users/root/home/.ssh directory. Although the distributed distfile.base.examples file has the rule to push these commented out you should replace the `id_rsa*` files with new keys by running `ssh-keygen -t rsa` in the directory, and edit `authorized_keys` replacing the distributed public key.
To check the distribution in for personal testing use:

```
svnadmin create /place/for/repository
cd repository
svn import -m "DACS 2.0 import" file:///place/for/repository
```

then use: `svn co file:///place/for/repository/Config/work` to get a working tree. Change into the working tree and run the `set_ignore` script from the distribution. This will set the `svn:ignore` properties on varios generated files. Running set_properties will set the `svn:owner`, `svn:group` and `svn:unix-mode` properties. Then do an
`svn ci -m "setting ignore and perms"` to commit the changes.
You will be able to browse files, check in changes and generate files. As root you can also check out a working copy into `/config` and use /config/Rdist to distribute files once you set up ssh access.

9.1.1 Subversion hook scripts

The subversion hook scripts are built for an svn+ssh access method and won't work well when the file access method is used. But you can check them out and look at them.
If you want to set this up for ssh access, you can move `/place/for/repository` to the home directory of your `dacsuser` in the repo subdirectory. You will need to remove the conf and hooks directory and check them out:

```
cd ~dacsuser/repo
rm -rf hooks
svn co file:///home/dacsuser/repo/SVN/hooks .
```

Similarly for the /place/for/repository/conf tree:

```
rm -rf conf
svn co file:///home/dacsuser/repo/SVN/conf .
```

9.1.2 Finish svn setup

Then recursively change the owner of the repository to dacsuser. Chmod 700 the dacuser/repository directory so only dacsuser and root can access the directory.
In the home directory of the dedicated dacsuser user, set up a dacsuser/.ssh/authorized_keys file so that it runs a forced svn command. There is an `authorized_keys.example` file supplied in the DACS release at repository/SVN/conf/authorized_keys.example. If it is moved to `authorized_keys` the `post-commit` hook script will copy that file into place so that all the changes to that file occur under subversion control. You should check out the file and put valid keys into it for root on the DACS masters and other authorized users of DACS.
The entry will look like:

```
no-agent-forwarding,no-X11-forwarding,no-port-forwarding,
command="/usr/bin/svnserve -t -r ~user --tunnel-user=fred"
ssh-rsa AAAA... ssh public key for fred ...== user_comment

no-agent-forwarding,no-X11-forwarding,no-port-forwarding,
command="/usr/bin/svnserve -t -r ~user --tunnel-user=admin"
ssh-rsa AAAA... ssh public key for admin ...== user_comment
```

This is split across multiple lines for display. In the authorized_keys file, all three displayed lines must be a single line. Also the path to /usr/bin/svnserv may be different on your machine. `~user` is the path to the parent directory of the owner of the repository.
There is one line for each user of the repository. For example the line with the "fred" tunnel user will display name `fred` in svn log as the author of changes and the name `fred` is used in the subversion access control file to authorize access to files in the repository.
An ssh public key can be created using ssh-keygen. The part of the file indicated by:

```
ssh-rsa AAAA... ssh public key for fred ...== user_comment
```

is the contents of the id_rsa file for that user. Each user should have a unique entry in the authorized keys file so that there is only one person who is able to authenticate. Without this it is impossible to assign changes to particular people and the auditing capability of the system is seriously impaired.
See the section "SSH configuration tricks" at
`http://svnbook.red-bean.com/en/1.1/ch06s03.html` in the subversion manual for details on setup.
If you deploy DACS remember you need to handle backups of the svn repository. See the subversion documentation for safe mechanisms (using hot-backup.py) to do this.

9.2 Setting up ssh access from the master to clients

The rdist version 6 program uses ssh to provide remote access to other systems. This section describes two different mechanisms for setting up ssh to permit rdist to access files and run remote commands. The first method is more restrictive than the second, but the second is more flexible for other administration tasks.
The DACS distribution file
`repository/Config/work/Config/distfile/distfile.base.example` includes two commented out rdist stanzas under the user label that push the `users/root/home` directory (which includes `.ssh/authorized_keys`) to all DACS hosts. It also has a second commented out stanza that pushes the root public and private ssh keys to DACS master and DACS standby setups. This allows DACS to verify and change the ssh keys that allow access to all it's managed hosts.
There are two ways to set up ssh access. The less secure and more useful, and the more secure and less useful. The less secure but more useful method will be discussed first because the DACS distribution implements this method.

9.2.1 Somewhat less secure

If you have been automating system administration chances are this is very close to what you already implement. There are two parts to this setup:

- set up each sshd_config file to permit only public key authentication but allow any command to be run.
- set up root/.ssh/authorized_keys to accept the public key from a limited set of hosts

The sshd_config file on your system (/etc/ssh/sshd_config or /etc/sshd_config most likely) should have the:

```
PermitRootLogin = without-password
```

This prevents password authentication and permits only public key authentication.
Root's .ssh/authorized_keys file should look like:

```
no-port-forwarding,no-agent-forwarding,no-X11-forwarding,
from="192.168.10.5,dacs.example.com,localhost,127.0.0.1"
ssh-rsa AAA... ssh public key ...==  root@dacsmaster
```

(all on one line) where the `from` value lists all the addresses/names of all the DACS master and standby (if used) hosts.
In order for an connection to occur:

1. the connecting ssh client must have the access to the private key that corresponds to the ssh public key.
2. the connection must come from one of the addresses in the from section

If a user uses `sudo ssh host` this setup will provide a shell for interactive use. Also with this setup, running `sudo ssh host ls` will perform a directory listing. When the rdist client connect's it runs `'ssh host rdist -S'` to spawn the server that it can talk to.

9.2.2 Most secure setup

There are two parts to this setup:

- set up the sshd_config file to permit root authentication via public key that runs a forced command (the rdist client program). This prevents the running of arbitrary commands as root.
- set up root/.ssh/authorized_keys to accept the public key from a limited set of hosts and run the rdist client as a forced command.

The sshd_config file on your system (/etc/ssh/sshd_config or /etc/sshd_config most likely) should have the setting:

```
PermitRootLogin = forced-commands-only
```

This prevents password authentication and permits only public key authentication but only for keys associated with a command.
Root's .ssh/authorized_keys file should look like:

```
no-port-forwarding,no-agent-forwarding,no-X11-forwarding,
from="192.168.10.5,dacs.example.com,localhost,127.0.0.1",
command="/usr/bin/rdistd -S"
ssh-rsa AAA... ssh public key ...==  root@dacsmaster
```

(all on one line) where the addresses in the from section are all possible address and names for the DACS master and standby (if used) hosts. The IP addresses allow access even if DNS is failing (in case you need to fix a broken /etc/resolv.conf via DACS). The `command` paramter specifies that this key can be used only to run the rdist command in server mode.
In order for a connection and update to occur:

1. the connecting ssh client must have access to the private key that corresponds to the ssh public key.
2. the connection must come from one of the addresses in the `from` section
3. the client must speak the rdist protocol to perform an update or remote command.

Although almost any change can be done using the rdist protocol, attempts to run commands from the DACS master host using: `sudo ssh host run this command` will fail. Also attempts to get an interactive shell using `sudo ssh host` from the DACS master server will fail.

9.3 Creating DACS working trees

Normally you create a working copy using:
`svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/work DACS`
where DACS is an arbitrary directory path. Note that you should not push files from this tree.

9.4 Creating DACS CCM master trees

A DACS CCM master tree is just a working copy of the DACS repository, except it's done as root and the path to the root of the working copy must be known to the Rdist script.
You create a normal working copy using:

```
svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/work DACS
```

where DACS is an arbitrary directory path. For creating master trees, do the same thing but as root:

```
sudo svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/work /config
```

This creates a DACS CCM master tree tied to the working tree that can update all hosts.
Using `/config` or `/DACS` as the master tree's root allows you to access all the hosts defined in the DACS database that are eligible for update. If you choose some other directory, you have to update the `%config_roots` associative array in the Rdist script.
If you want to use split production/test trees check out a production tree using:

```
sudo svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/production
/config
```

and a test tree to:

```
sudo svn co svn+ssh://dacsuser@dacsmaster:/repo/Config/work /config.test
```

Then the changes that are done to a working tree will show up when /config.test/Rdist is run to allow pushing the changes to the set of hosts tagged as providing the service TESTHOST. Again /config.test is a recognized root that permits only a subset of the available hosts to be updated. The working tree should be the only tree that people check out and work in. All changes to the production tree should be copied from entries in the checked in working tree. I do not believe that svn can enforce this. However it should record an audit trail that will show when this has been violated if you use the DACS recommended promotion mechanism.
Also you can check out partial DACS CCM master trees. This can be useful when you have a lot file delegation as it provides a limited subset of files available for a user to push. To set up a partial master tree execute:

```
sudo svn co -n svn+ssh://dacsuser@dacsmaster:/repo/Config/work /partial.tree
sudo svn up  /partial.tree/Config
sudo svn up /partial.tree/other_dir
```

and repeat for all the directories you want. If you want to use this method, note that the Distfile generation mechanism requires a few changes as well to remove references to the missing directories. Generally this can be done using a perl or sed script to remove the entries from the filepp expanded distfile.base.

9.5 Replicating (synchronizing) DACS svn master repository

If you are using subversion 1.4 or newer, you can use svnsync to maintain a replica of the svn repository. The distribution includes subversion hook files that will synchronize a DACS master svn repository to replica repositories for redundancy.
To set up replication over ssh, create ssh keys for the DACS subverion repository owner (called dacsuser in the distribution) on the master server. Use `sudo -u dacsuser ssh-keygen -t rsa` on the master host to place the files in dacsuser/.ssh/id_rsa and id_rsa.pub. (Note you can also set these keys up in DACS so that they are installed on all the replicas as well in case you have to make a replica into the master svn tree. See how the users/root/home directory is pushed to the DACS_HOSTS and set up something similar for your dacsuser.) Add the id_rsa.pub to the dacsuser's authorized keys file by editing the SVN/conf/authorized_keys file installed as part of the repository. See the "Importing the DACS repository tree" section above for more details.
Once the key is installed, we can set up the replica repositories using the user `dacsuser` with the repository in the `~dacsuser/repo` directory. First in the DACS database:

- add SVNDACS_STANDBY to the host(s) running the replicas and check-in the change

then

- manually or using DACS set up the `dacsuser` account
- copy the existing dacsuser/.ssh/* files on the master to the replicas dacsuser/.ssh/ directory
- chown -R dacsuser dacsuser on the replicas

On the DACS subversion master, dump the unique universal identifier (uuid) using:

- svnadmin dump dacsuser/repo -r 0 > uuid

This allows `svn switch --relocate` (discussed below) on a checked out DACS tree to work when swapping to the replica repository.
This part is done as the user dacsuser on the replicas:

- svnadmin create dacsuser/repo
- svnadmin load dacsuser/repo < uuid
  where uuid is the file generated above.

- chmod 700 dacsuser
  to protect the information in the repository

- ln -s /bin/true dacsuser/repo/hooks/pre-revprop-change
  this permits the sync operations from the master to work. It will be replaced with the normal hook scripts when the inital synchronization is done. The normal hook scripts could be used but they make the initial synchronization take longer.

Then on the dacsmaster host where the svn repository is located start the replication process:

- sudo -u dacsuser env -i /usr/bin/svnsync init –non-interactive
  svn+ssh://dacsuser@replica1/repo
  file:///home/dacsuser/repo

This initializes revision 0 of the replica repository with the properties needed to make svnsync work. The `env -i` command clears the environment of the svnsync process. This makes ssh use the `~dacsuser/.ssh/id_rsa` key for access even if there is an ssh-agent running. Now start the sync process from the dacs master as the dacsuser. This replays every commit to the master repository on the replica and will take a while for large repositories:

- sudo -u dacsuser env -i /usr/bin/svnsync sync –non-interactive
  svn+ssh://dacsuser@replica1/repo

When that completes, check out the standard hook and configuration files on the replica server. On the replica server as the dacsuser run:

- rm -rf dacsuser/repo/conf

- rm -rf dacsuser/repo/hooks

- svn co file:///home/dacsuser/repo/SVN/conf
  /home/dacsuser/repo/conf

- svn co file:///home/dacsuser/repo/SVN/hooks
  /home/dacsuser/repo/hooks

This deploys the hooks and conf directories and they will be automatically maintained from now on by the post-commit script. Note that the automatic maintainance does not validate the scripts, it merely does an `svn update`, so changing the files directly in the hooks directory can cause problems if the changes cause the scripts to become invalid after an update.
Copy the file dacsuser/repo/commit-log from the master to the same place on the slave. This file logs all the commits to the repo.
All replicas should be locked using:

```
echo "repository is replica" | sudo tee ~dacsuser/repo/conf/LOCKED
```

The existance of the LOCKED file prevents updates to the repository unless the author of the changes is the repository owner which is used ONLY to commit replication information to the repository.
Note that `Rdist users` will push the LOCKED file to all replicas and will warn when a master host has a locked file.

9.5.1 Configuring the replicas in SVN

You need to create the slave_urls file in the subversion conf directory (a sample file is provided in the distribution as slave_urls.example).
Populate that file with the svn+ssh url's used for the svnsync operations above. E.G. `svn+ssh://dacsuser@replica1/repo` where dacsuser and replica1 are replaced with a real user and host on your network. There should be one url per line in the file.

9.5.2 Using replication

Replication should be pretty automatic once it is set up. Root will receive email if there are replication errors (so you want to forward root email on your DACS subversion master to somebody who will read it).
The hook scripts recognize when the repository access is done by the repository owner (dacsuser in the distribution) and will update the operational files on the replicas to match the master. So ssh access (if you use the config/authorized_keys file), authorization and hook scripts will be identical on the replicas and the master.
Check-ins not done as part of replication are done only on the master and trigger email notifcations and replication operations to the replicas.

9.5.2.1 Switching masters (replica becomes a master)

When you lose the DACS subversion server, you need to relocate the DACS master tree to the replica server. Change to the root of the DACS master tree and run:

```
sudo env -i /usr/bin/svn switch --relocate \
    svn+ssh://dacsuser@dacsmaster svn+ssh://dacsuser@replica1
```

replacing `dacsmaster` and `replica1` with real host names.
Now when Rdist runs it will sync from the replica repository rather than the master repository.
Now to enable check-ins to the replica, you need to remove the file `~dacsuser/repo/config/LOCKED`. Also you should move the SVNDACS_MASTER service from the dead master to the replica you are using as a new master and remove the SVNDACS_STANDBY token from the replica. After this change running `Rdist users` will warn you if you forgot to unlock the master repository.
The command `svn switch --relocate ....` needs to be used on any DACS working copy that wishes to access (e.g. for checking in changes) the new repository.
You should also update the slave_urls's file and remove the url for the replica that is now the master. The new master will try to update itself, and this will fail generating an email to root. Exiting the slave_urls file will prevent this email.

9.5.2.2 Moving masters

To move back to the original master, delete it and set it up as a replica of the current master then execute the steps in "Switching masters" to make it a new master.
You can use svnsync and other commands to bring it back in sync but starting over from scratch is the most foolproof method.

9.6 Other documentation

The table of contents for the automatically generated documention (from robodoc comments) for the Config tree is located in the release repository directory under Config/docs/toc_index.html. That page also provides links to the other documentation indexes.
For access to the dacs defined classes documentation see Config/docs/bin/dbreport.html.

Chapter 10

Glossary

CCM acronym for Computer Configuration Management cf. SCM

CNCM acronym for Computer and Network Configuration Management. Another name for CCM cf. SCM

CCM (repository) tree see master tree or repository tree.

class see class variable (rdist) or class macro (filepp)

class variable (rdist) This is just a normal rdist variable that has contents generated by the DACS class mechanism. It is a space separated list of hostnames that all match some specific criteria.

class macro (filepp) A space separated list of hostnames all matching some criteria. The naming matches that for "class varibles (rdist)" but with `FILEPP_` prepended to the rdist class name.

client see client host

client host a system that has files or services managed by DACS.

CMDB configuration management database

DACS CCM tree see master tree or repository tree.

DACS server the system or systems that are allowed root access to a client host to update and manage files on the client.

dacsmaster a hypothetical host that has the dacsuser account and provides access via ssh to the DACS subversion repository.

dacsuser a hypothetical account that provides access via ssh to the subversion repository for DACS.

implementer person responsible for defining new system configurations in DACS. This person defines new tokens in the database and maps files to those configurations.

label a mechanism to allow selection of rdist stanzas in a Distfile. It is specified on the rdist or Rdist command line: `Rdist label1 label2...`

master repository a VCS repository that allows commits from users and is used to update a master tree.

master tree a copy of the VCS tree done as root that will not have any local modifications done to it. It will not be used for checkins to the VCS. It is used as the master tree from which files are distributed to client machines. Compare to 'working copy'.

replica repository a VCS repository that receives updates from a master repository and can be used as a master repository if the master repository dies.

repository tree a generic term for the set of files in a working copy or master tree.

slave repository see replica repository

SCM software configuration management cf. CCM

svn the command used to invoke the subversion VCS. Also used as an abbreviation for subversion.

ticket trouble ticket/work order/change request ...

target see target (make) or labels if you are discussing rdist

target (make) the item that is to be made by the make system. It is specified on the make command line: `make target`.

VCS version control system. Either CVS (concurrent versioning system) or subversion.

user a person who interacts with DACS by running dbreport, Rdist and changing the files that exist under DACS control.

working copy a copy of the VCS repository checked out as a local user and created for the purpose of performing updates to files and checking those changes into the VCS.

Index

Index