Coda File System
User and System Administrators Manual

M. Satyanarayanan, Maria R. Ebling, Joshua Raiff, Peter J. Braam

v1.1, 20 August 1997

The *Coda File System* is a descendant of the *Andrew File System*. Like AFS, Coda offers location-transparent access to a shared Unix file namespace that is mapped on to a collection of dedicated file servers. But Coda represents a substantial improvement over AFS because it offers considerably higher availability in the face of server and network failures. The improvement in availability is achieved using the complementary techniques of *server replication* and *disconnected operation*. Disconnected operation proven especially valuable in supporting portable computers. This document is a reference manual for Coda users and system administrators.

Contents

Chapter 1

Preface

Welcome to Coda! Whether you are at Carnegie Mellon or elsewhere, we hope that this manual will help you make effective use of the Coda File System. The manual is written from two different perspectives: end user and system administrator. Chapters 2 () through 4 () are intended for end users, while Chapters 3 () through 13 () are intended for system administrators. Also, 14 () gives an overview of the Coda source tree layout. The Unix-style man pages in Appendix E () provide the gory details of Coda commands and file formats.

In writing this manual, we have assumed that you are an experienced Unix user, familiar with the broad concepts of distributed file systems. If you are a system administrator, we assume that you are familiar with administering Unix systems. You will find Coda especially easy to use if you are already fluent in using AFS.

We have also assumed that you are familiar with the design goals and architecture of Coda. The best way to learn about these is to read the overview and design rationale papers on Coda [**?**], [**?**], [**?**]. We urge you to obtain copies of these papers and to read them before attempting to use Coda. Those papers provide context and information upon which this manual relies and does not repeat.

Since Coda is an experimental system and not a commercial product, you will inevitably encounter rough edges. Within the limits of our resources, we will do our best to fix these problems and to improve the system over time. But we do ask that you try to characterize the problem as accurately as possible, and to try and obtain a repeatable and concise instance of it.

We encourage you to talk to other Coda users and to share your experience. If you are at CMU-SCS, you should subscribe to the Coda bulletin boards. The three most important ones for you as an end user are `cmu.cs.proj.coda.{general,laptops,bugs}`. Report bugs that you find to the latter bboard. System administrators should also subscribe to the `cmu.cs.proj.coda.{changelog,backuplogs}`.

After you have taken the time to learn the system well, please give us your feedback. We would like to improve the system and this manual.

1.1 Further Reading

Besides the three papers mentioned above, there are a number of papers that address specific aspects of Coda. These include:

- A description of the MiniCache, which allows the client manager to reside outside the kernel without excessive loss of performance [**?**].
- A description and performance analysis of the technique used in Coda for transparent directory resolution [**?**].
- A manual on the remote procedure call, *RPC2,* and threading, *LWP,* mechanisms used in Coda [**?**].
- A paper and manual on the transactional facility, *RVM,* used on Coda clients and servers [**?**], [**?**].
- A detailed description of support for disconnected operation in Coda [**?**]. Chapter 4 of this document offers an excellent overview of the implementation structure of clients and servers.
- A discussion of relevant security issues and mechanisms to address them [**?**]. This paper discusses security in AFS-2, but the mechanisms in Coda are virtually identical.

1.2 Acknowledgements

Coda is the work of many individuals. Contributors to the design and implementation of various aspects of the system include: Jay Kistler, Puneet Kumar, David Steere, Lily Mummert, Maria Ebling, M. Satyanarayanan, Hank Mashburn, Brian Noble, Lu Qi, Josh Raiff, Ellen Siegel, Anders Klemets, and Kudo Masahi. Many of these individuals have also contributed to the writing of this manual. The system has improved considerably in response to feedback from users outside the Coda project. The earliest of these users (Tom Mitchell, Manuela Veloso, and Matt Zekauskas) deserve special thanks for their willingness to sail into uncharted waters!

1.3 Reading List

Chapter 2

Getting Started

The Coda file system expands the local file system hierarchy to include shared files much like the *Sun Network File System* and the *Andrew File System*. Figure 2.1 (XXX) below shows the users view of the file system hierarchy on a machine with all three types of file systems.

Data sharing in Coda, like that in Andrew, is supported by a distributed file system that appears as a single large subtree of the local file system on each workstation. This subtree is identical on all workstations. Thus, processes on two different workstations can read and write files in this subtree just as if they were running on a single timesharing system with the difference that processes on the different workstations see changes only when files are opened and closed. Familiarity with AFS is helpful when learning about Coda.

This chapter introduces essential Coda commands. For more details on these commands, refer to their man pages.

2.1 Authentication

Once you are logged in to your workstation, you need to get a Coda authentication token by running the **clog** program. **clog** will prompt you for your Coda password and use it to get a token from the *authentication server*. This token will expire in about 25 hours. After the token expires, you must use **clog** to authenticate

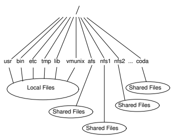

Figure 2.1: Coda filesystem view at a workstation

for another 25 hours. The following in an example of running **clog** twice. The first time, the wrong password was entered:

```
% clog
Password:
Invalid login (RPC2_NOTAUTHENTICATED (F))
% clog
Password:
%
```

To see your newly acquired token, use **ctokens**. This will display the tokens and their expiration time for your UID.

```
% ctokens

Tokens held by the Cache Manager:

UID=2534 : [Expires Mar 19 10:50]
```

Use the **cpasswd** command to change your Coda password. As with **passwd**, **cpasswd** will prompt for your current password, then ask you to enter a new password twice.

```
% cpasswd
Changing password for raiff
Old password:
New password for raiff:
Retype new password:
Password changed, it will be in effect in about 1 hour
```

You can "log out" of Coda by using the **cunlog** command to tell *venus* to forget your tokens. Once you run **unlog**, you will have the same privileges as an anonymous Coda user until you acquire a new authentication token.

2.2 Coda File Protection

Coda provides a close approximation to UNIX protection semantics. An *access control list* (ACL) controls access to directories by granting and restricting the rights of users or groups of users. An entry in an access list maps a member of the protection domain into a set of rights. User rights are determined by the rights of all of the groups that he or she is either a direct or inderect member. In addition to the Coda access lists, the three owner bits of the file mode are used to indicate readability, writability, and executability. You should use *chmod*(1) to set the permissions on individual files. Coda rights are given as a combination of **rlidwka** where:

- **r** *Read*. With this right, you can read any file in the directory.

- **l** *Lookup*. Lookup allows you to obtain status information about the files in the directory. An example is to list the directory contents.
- **i** *Insert*. Allows the user to create new files or subdirectories in the directory.
- **d** *Delete*. Allows the user to remove files or subdirectories.
- **w** *Write*. Allows the user to overwrite existing files in the directory.
- **k** *Lock*. The lock right is obsolete and only maintained for historical reasons.
- **a** *Administer*. Allows the user to change the directorys access control list.

Coda also has *negative* rights, which deny access. Any of the normal rights listed above can also be negative.

Access control lists are managed with the **cfs** command through the **listacl** and **setacl** options. These commands can be abbreviated as **la** and **sa** respectively. To see the access control list of any directory in a Coda file system, use **cfs la**. The following example displays the current directorys ACL:

```
% cfs la .
      System:AnyUser  rl
               raiff  rlidwka
```

The displayed list, shows that the user "raiff" has all of the access rights possible on the directory and that the group "System:AnyUser" has read and lookup privileges. System:AnyUser is a special Coda group that includes all users.

A second example shows another group, System:coda. Anyone who is a member of the group, will have the groups access rights:

```
% cfs la /coda
         System:coda  rlidwka
      System:AnyUser  rl
```

Use **cfs sa** to change or set a directorys access control list. Options to **cfs sa** include *-negative* to assign negative rights to a user and *-clear* to clear the access list completely before setting any new access rights. You can also use *all* or *none* to specify all rights or no rights respectively. To remove System:AnyUsers access to the current directory, you would issue the following command:

```
% cfs sa . System:AnyUser none
```

To give System:AnyUser read and lookup rights, use:

```
% cfs sa . System:AnyUser rl
```

To deny rights to a user, use the **-negative** switch:

```
% cfs sa -negative . baduser rl
```

This will deny "baduser" read and lookup rights, even though any other user has these rights. Note that negative rights are maintained separately from the normal rights, so to re-instate badusers read and lookup access, you must use:

```
% cfs sa -negative . baduser none
```

If you omit the *-negative* switch, then "baduser" will still be denied read and lookup access.

2.3 Disconnected Operation

If all of the servers that an object resides on become inaccessable, then the client will use the cached copy of the object (if present) as a valid replica. When the client does this, it is operating in *disconnected mode*.

Disconnected mode may be the result of a network failure, or it could be the result of removing a laptop from the network. If you make sure all of the files you want to use are cached on your laptop, you can travel with it and access your files as if you were still on the network.

Unfortunately, a cache miss while operating in disconnected mode is not maskable, and you will get a *connection timed out* error message. Coda allows you to mark or *hoard* files with caching priorities to help keep the ones you want in the cache.

When you are in disconnected mode, you may want to checkpoint the modify log that Coda keeps of which directories have changed. Use **cfs checkpointml** to do this. Checkpointing the modify log will ensure that changes you have made will not be lost if the system crashes. Coda uses this modify log when it reintegrates with the servers.

Coda also supports low bandwidth access over SLIP. You can use this to periodically reintegrate and cache new files when you are on a trip.

When you reintegrate after operating in disconnected mode, watch your **codacon** or run the command:

```
% tail -f /usr/coda/etc/console
```

This file will let you know if the reintegration was successful. If it was not, then the files that you modified will be put in a tar file in */usr/coda/spool/<uid>*. Reintegration fails, for example, when you modified a file in disconnected mode and someone else also modified that file on the servers. Section 3.3 (XXX) has more on reintegration.

2.4 Hoarding

Coda allows you to advise the cache manager, *venus*, of critical files that it should try to keep in the cache. You indicate the relative importance of the files by assigning priorities to them. This is known as *hoarding*. Venus maintains an internal *hoard database* of these files. Hoarding a file helps to ensure that it will be available when operating in disconnected mode. See the **hoard**(1) man page for information on hoard files and sections 3.1 (XXX) and 3.2 (XXX) of this document for an example of how to set up your hoard database. A convenient way of setting up your hoard database is by creating a file with commands for the **hoard** program. This file is known as a *hoard file*.

2.5 Repairing Conflicts

As a result of Codas optimistic replica management, object replicas can conflict on different servers. A conflict arises when the same object is updated in different partitions of a network. For instance, suppose a file is replicated at two sites (say, serverA and serverB). If these two sites become partitioned and a user on each side of the partition updates the file (userA updates the file on serverA while userB updates the file on serverB), the file will be in conflict when the partition ends. Conflicts may also arise at the end of disconnected operation.

Coda guarantees conflict detection at the first request for that object when both servers are accessible. When a conflict is detected, Coda attempts to perform automatic conflict resolution. In simple cases, the conflict will be resolved automatically, a process which is transparent to the user except for a time delay in accessing the object. However, in more difficult cases, automatic conflict resolution fails and the object is marked *in conflict*. File system calls on an object which is in conflict fail with the same error code as if the object were a dangling, read-only symbolic link (usually, ENOENT). The conflict must be resolved by a user with appropriate access to the object. To help users resolve conflicts, Coda provides a repair tool which is discussed in Section 3.6 (XXX).

Chapter 3

Common Scenarios

There are several common scenarios that you may encounter. This chapter attempts to list as many of them as possible and suggest how to handle each scenario.

3.1 Constructing a hoardfile

Coda allows you to give files priorities that affect the cache manager. The higher the priority, the lower the possibility that a file will get flushed from the cache to make space for a another file. These priorities are stored in a hoard database internal to Venus. This database is preserved across invocations of Venus, but will be erased when Venus is re-initialized.

The best way to set up a hoard database is by creating hoard files. After you've created the files once, you do not need to do it again for that set of files. You can create a hoard file by hand or by using the *spy* program. See the **hoard**(1) man page for more details.

To run *spy*:

1. Run spy in the background, redirecting its output to a file.
2. Run all of the programs and access files you want to hoard.
3. Send a SIGTERM to spy (Do not use ˆC)
4. Sort the output file, removing duplicates
5. Remove unnecessary entries
6. On each line add `a` at the beginning and a priority at the end

The following is an example of creating a hoard file for gnu-emacs. Note that while running gnu-emacs, I explicitly enter "scribe mode." This makes sure that the scribe-specific files are fetched.

```
% spy > gemacs.out &
[1] 316
% gnu-emacs
```

```
% kill %1
%
[1]    Done                 spy > gemacs.out
sort -u gemacs.out > gemacs.hdb
% cat gemacs.hdb
/coda
/coda/i386_mach/omega/usr/local/emacs
/coda/i386_mach/omega/usr/misc/.gnu-emacs
/coda/misc/gnu-emacs/i386_mach/omega/bin/gnu-emacs
/coda/misc/gnu-emacs/i386_mach/omega/lisp/scribe.elc
/coda/misc/gnu-emacs/i386_mach/omega/lisp/term/x-win.el
/coda/misc/gnu-emacs/i386_mach/omega/lisp/x-mouse.elc
/coda/usr
```

Next I would delete the first and last line of the file as I do not need them. Then add the **hoard** specific commands. The final file looks like:

```
a /coda/i386_mach/omega/usr/local/emacs 600
a /coda/i386_mach/omega/usr/misc/.gnu-emacs 600
a /coda/misc/gnu-emacs/i386_mach/omega/bin/gnu-emacs 600
a /coda/misc/gnu-emacs/i386_mach/omega/lisp/scribe.elc 600
a /coda/misc/gnu-emacs/i386_mach/omega/lisp/term/x-win.el 600
a /coda/misc/gnu-emacs/i386_mach/omega/lisp/x-mouse.elc 600
```

The `a` that starts each line tells hoard to "add" the named file to the database. The 600 that ends each line sets the file's priority. You may also specify additional attributes for each line. These attributes are separated from the priority by a ":" and are:

- **c** Current children of the given directory will inherit its hoard status.
- **c+** Current and future children of the given directory will inherit its hoard status.
- **d** Current descendents of the given directory will inherit its hoard status.
- **d+** Current and future descendents of the given directory will inherit its hoard status.

For example, to hoard all of the emacs directory, its descendents and any future descendents, I would include the following line in a hoardfile:

```
a /coda/i386_mach/omega/usr/local/emacs 600:d+
```

This ensures you get all of the files you need, but you will use tens of megabytes of cache space to hoard many files that you do not need, so oftern you want to be more specific with respect to which files to hoard.

Other valid command to *hoard* are clear, delete, list, and modify. See the *hoard*(1) man page for more details on these commands.

3.2 Hoarding for a Weekend

One of the most common uses of a Coda laptop is to take it home overnight or for the weekend. Naturally, you want to be sure that all of the files that you need over the weekend are in the cache; otherwise, there is little point in bringing the laptop home. The *hoard*(1) program helps you do this. Create a hoard file, as described in Section 3.1 (XXX), for each project you want to work on. You may also want hoard files for your personal files, such as your home directory if its in Coda, and other tools that you plan on using. Its best to clear the hoard database whenever you switch projects to make sure you have enough space in your cache. You might consider having *clear* as the first command in your personal hoard file. If you do, make sure you always run hoard with this file before any other files. Once youve built hoard files for all of your tools and projects, its simply a matter of running hoard to build the hoard database you want. When you run *hoard*, you must be logged onto the machines console, (dont run X). About fifteen minutes before you are ready to leave, force a hoard walk with the following command:

```
% hoard walk
```

This will cause *venus* to attempt to cache all of the files in the hoard database. *Wait until the hoard command completes*. You are now ready to disconnect from the network. You are encouraged to try all of the commands you intend on using after you disconnect. If you are missing some files, it will be easy to reconnect and hoard them.

3.3 Reintegrating After a Disconnected Session

When you reconnect to the network after a disconnected session, Coda will automatically try to reintegrate your changes with the Coda servers. You must be authenticated before reintegration occurs. Watch the file */usr/coda/etc/console* with the **codacon** command or by running: **tail -f /usr/coda/etc/console**. The reintegrations status will be written to this file.

If the reintegration was successful, the log entries would look like:

```
Reintegrate u.raiff, (1, 244) ( 13:33:43 )
coda: Committing CML for u.raiff ( 13:33:43 )
coda: Reintegrate: u.raiff, result = SUCCESS, elapsed = 2640.0 (15.0, 2609.0,
15.0) ( 13:33:43 )
coda:   delta_vm = 0, old stats = [0, 1, 0, 0, 0] ( 13:33:43 )
coda:   new stats = [   0,   0.0,     0.0,    1,   0.2], [   0,   0.0, 0.0,
 0,   0.0] ( 13:33:43 )
```

The following example is from a failed reintegration on the volume u.raiff.

```
Reintegrate u.raiff, (1, 244) ( 13:27:10 )
coda: Checkpointing u.raiff ( 13:27:10 )coda: to /usr/coda/spool/2534/u.raiff@@%
coda%usr%raiff ( 13:27:10 )
coda: Aborting CML for u.raiff ( 13:27:10 )
coda: Reintegrate: u.raiff, result = 198, elapsed = 2437.0 (15.0, 2265.0, 531.0)  ( 13:27:10 )
coda:   delta_vm = 1000, old stats = [0, 0, 1, 0, 0] ( 13:27:10 )
```

```
coda:   new stats = [   0,   0.0,     0.0,    1,   0.2], [   0,   0.0, 0.0,   0,
   0.0] ( 13:27:10 )
```

Notice that the *change modify log* (CML) was checkpointed to */usr/coda/spool/2534/u.raiff@@%coda%usr%raiff* . This file is a tar file containing the changes that were made on during the disconnected session. The files in the tar file are relative to the root of u.raiff. The **cfs examineclosure** and **cfs replayclosure** will show which files were not reintegrated and force a reintegration respectively.

3.4 Dealing With a Flaky Network

When the network is acting up, you can use Coda to help isolate yourself from the networking problems. Set up your hoard database so that Venus will hoard the files you are working on. Then, disconnect from the Coda servers with the **cfs disconnect** command. To Coda, this is equivalent to physically disconnecting from the network.

Once the network becomes stable, you can use **cfs reconnect** to reconnect yourself to the Coda servers and re-integrate your work. Dont forget to clear your hoard database with **hoard clear** once you are done working on the set of files that you hoarded.

Note: AFS will not be affected by cfs, so access to AFS files will still be affected by the network problems.

3.5 Reintegrating Over the Phone Line

If you are planning on taking a Coda laptop on an extended trip, you should consider using SLIP to reinitgrate with the Coda servers periodically. Using SLIP will allow updates to be visible to other Coda users, protect against client crashes such as hard drive failure or theft, and allow you to work on multiple projects, even when your cache space is not large enough for all of the projects. By using the following instructions, you can reintegrate over the phone and change the files in your hoard database.

1. Read the *dialup* manual page, and */afs/cs/help/03-Communication/03-Tcons_and_Dialups/slip.doc* */afs/cs/help/03-Communication/03-Tcons_and_Dialups/cisco_tcon.doc*

2. Get an account on a terminal server from facilities. This server is the name that you will use when you start slip.

3. Connect to the terminal server and start slip.

4. Run `/etc/slattach /dev/com{0,1}` speed. If you are using the internal modem, specify /dev/com0 and 2400. If you are using an external modem, use /dev/com1 and whatever speed (9600).

5. Exit your communications program (kermit, whatver). slattach holds the line open.

6. Now you have to reset the routes in your routing table. First delete the old routes:

```
% set slipaddr = 128.2.254.129 # address of ts2.srv.cs.cmu.edu
% set gwaddr = 128.2.254.36    # address of gw.cs.cmu.edu

% /etc/route delete 0 $gwaddr
% /etc/route delete net 128.2 $hostaddr
% /etc/route -f delete $hostaddr $hostaddr
```

and config the slip interface up with the new routes:

```
% /etc/ifconfig sl0 $hostaddr $slipaddr -trailers up
% /etc/route add net 128.2 $slipaddr 0
% /etc/route add 0 $slipaddr 1
```

If youve started up disconnected, you will also have to run the command:

```
% ifconfig par0 down
```

Finally, tell Coda to see which servers it can communicate with:

```
% cfs checkservers
```

Your laptop will now behave as if it is on the network. Response time to commands will be sluggish.

If you want to stop running SLIP before you shut down your computer, simply turning off your modem or killing *slattach* will terminate your SLIP connection.

3.6 Repairing an Inconsistent Directory

Occasionally, a directory entry will become inconsistent. This happens when there is a conflict between file server replicas that Coda cannot automatically resolve or a reintegration failed because of a local update the conflicts with the global state. The most common causes of a conflict are when the file servers are partitioned and a file is changed on more than one of the partitions or when a disconnected client updates a file that is also updated on the servers. When this happens, the directory containting the conflict will now look like a symbolic link and will be pointing to its *fid*. For example, if a directory, *conflict*, is inconsistent, it will now appear as:

```
% ls -l conflict
lr--r--r--  1 root      27 Mar 23 14:52 conflict -> @@7f0000b3.00000005.0000011a
```

Most applications will return the error File not found when they try to open a file that is inconsistent. You need to resolve this conflict by using the *repair*(1) tool.

3.6.1 Server/Server Conflicts

Once you run repair, you need to do a "beginRepair" on the object that is inconsistent. After "beginRepair" is issued, the inconsistent directory will have an entry for each of the replicated volumes. You can look at all of these to decide which copy you want. Use repair to copy the correct version and clear the inconsistency. In the following example the file *conflict/example* is replicated on three servers. It has gone inconsistent.

```
% ls -lL conflict
lr--r--r--  1 root          27 Dec 20 13:12 conflict -> @@7f0002ec.000000e3.000005d1
% repair
The repair tool can be used to manually repair files and directories
that have diverging replicas.  You will first need to do a "beginRepair"
which will expose the replicas of the inconsistent object as its children.


If you are repairing a directory, you will probably use the "compareDir" and "doRepair" commands.

For inconsistent files you will only need to use the "doRepair" command.

If you want to REMOVE an inconsistent object, use the "removeInc" command.

Help on individual commands can also be obtained using the "help" facility.
* begin conflict
a server-server-conflict repair session started
use the following commands to repair the conflict:
        comparedirs
        removeinc
        dorepair
* ^Z
Stopped
% ls conflict
gershwin.coda.cs.cmu.edu        schumann.coda.cs.cmu.edu
% ls conflict/*
conflict/gershwin.coda.cs.cmu.edu:
example

conflict/schumann.coda.cs.cmu.edu:
example
% fg
repair
compare
Pathname of Object in conflict?  [conflict]
Pathname of repair file produced?  []  /tmp/fix


NAME/NAME CONFLICT EXISTS FOR example

-rw-r--r--  1 raiff          0 Dec 20 13:10 gershwin.coda.cs.cmu.edu/example
-rw-r--r--  1 -101           0 Dec 20 13:11 schumann.coda.cs.cmu.edu/example


/coda/project/coda/demo/basic/rep/conflict/gershwin.coda.cs.cmu.edu/example
        Fid: (0xb0.612) VV:(0 2 0 0 0 0 0 0)(0x8002f23e.30c6e9aa)
/coda/project/coda/demo/basic/rep/conflict/schumann.coda.cs.cmu.edu/example
```

```
        Fid: (0x9e.5ea) VV:(2 0 0 0 0 0 0 0)(0x8002ce17.30d56fb9)
Should /coda/project/coda/demo/basic/rep/conflict/gershwin.coda.cs.cmu.edu/example be removed?   [no]  y
Should /coda/project/coda/demo/basic/rep/conflict/schumann.coda.cs.cmu.edu/example be removed?   [no]
Do you want to repair the name/name conflicts  [yes]
Operations to resolve conflicts are in /tmp/fix
* do
Pathname of object in conflict?  [conflict]
Pathname of fix file?  [/tmp/fix]
OK to repair "conflict" by fixfile "/tmp/fix"?  [no]  yes
SCHUMANN.CODA.CS.CMU.EDU  succeeded
GERSHWIN.CODA.CS.CMU.EDU  succeeded
* quit
% ls conflict
example
% exit
```

3.6.2 Local/Global Conflicts

Local/global conflicts are caused by reintegration failures, which means that the mutations performed while the client was disconnected are in conflict with the mutations performed on the servers from other clients during the disconnection. The objects involved in local/global conflict are represented in the same fashion as server/server conflicts, i.e., they become dangling symbolic links.

To start a local/global repair session for an object OBJ, you need to invoke the repair tool and issue the "beginrepair" command with the pathname of OBJ as the argument. Once the repair session is started, both the local and global replicas of OBJ are visible at OBJ/local (read-only) and OBJ/global (mutable and serving as the workspace for storing the repair result for OBJ and its descendants). The central process of repairing the local/global conflicts on OBJ is to iterate the local-mutations-list containing all the local updates performed on OBJ or its descendants, which can be displayed by the "listlocal" command. Each operation in the list must be accounted for and the repair tool cooperates with Venus to maintain the current-mutation being iterated. The "checklocal" command can be used to show the conflict information between the current-mutation and the global server state. You can advance the iteration to the next operation using either the "preservelocal" or the "discardlocal" command with the former replaying the current-mutation operation on the relevant global replicas. You can also use the "preservealllocal" and "discardalllocal" commands to speed up the iteration. Because the global replica OBJ is mutable, existing tools such as "emacs" etc. can be directly used to make the necessary updates. The "quit" command is used to either commit or abort the repair session. The man page on on the repair commands contains more detailed information, and the following simple example illustrates the main process of repairing a local/global conflict.

Suppose that during disconnection, a user creates a new directory */coda/usr/luqi/papers/cscw/figs* and stores a new version for file */coda/usr/luqi/papers/cscw/paper.tex*. However, during the disconnection his co-author also creates a directory */coda/usr/luqi/papers/cscw/figs* and stores some PS files in it. Upon reintegration a local/global conflict is detected at */coda/usr/luqi/papers/cscw*.

```
% ls -l /coda/usr/luqi/papers/cscw would show
lr--r--r--  1 root          27 Dec 20 00:36 cscw -> @@7f000279.00000df3.0001f027
% repair
```

```
* begin
Pathname of object in conflict?  []  /coda/usr/luqi/papers/cscw
a local-global-conflict repair session started
the conflict is caused by a reintegration failure
use the following commands to repair the conflict:
        checklocal
        listlocal
        preservelocal
        preservealllocal
        discardlocal
        discardalllocal
        setglobalview
        setmixedview
        setlocalview
a list of local mutations is available in the .cml file in the coda spool directory

* !ls -l /coda/usr/luqi/papers/cscw
total 4
drwxr-xr-x  3 luqi         2048 Dec 20 00:51 global
drwxr-xr-x  3 luqi         2048 Dec 20 00:51 local
Back to *

* listlocal
local mutations are:

Mkdir   /coda/usr/luqi/papers/cscw/local/figs
Store   /coda/usr/luqi/papers/cscw/local/paper.tex (length = 19603)

* checklocal
local mutation: mkdir /coda/usr/luqi/papers/cscw/local/figs
conflict: target /coda/usr/luqi/papers/cscw/global/figs exist on servers

* discardlocal
discard local mutation mkdir /coda/usr/luqi/papers/cscw/local/figs

* checklocal
local mutation: store /coda/usr/luqi/papers/cscw/local/paper.tex
no conflict

* preservelocal
store /coda/usr/luqi/papers/cscw/global/paper.tex succeeded

* checklocal
all local mutations processed

* quit
commit the local/global repair session?  [yes]
```

Chapter 4

Troubleshooting

This chapter is meant to help solve some of the problems you may run into when using Coda.

4.1 File System Full

You may get a message of the form, */usr: file system full* when you have not put any extra data on that disk. If the venus.cache directory is on that disk, that is probably your problem. You can execute either *cfs flushobject* <*object*> or *cfs flushvolume* <*dir*> to try to free up some disk space. **Be careful when flushing anything while disconnected. You will lose any changes that you made**. In the long term, the cache size in */usr/coda/etc/vstab* is proabably too large for the disk and should be made smaller.

Chapter 5

System Overview

Each Coda client sees the Coda File System as a single tree, /coda. In reality this tree is an illusion supported by *system control machine* (SCM), several dedicated file servers, and a local area network. One of the servers may double as the (SCM). Figure 5.1 (XXX) illustrates the server organization.

5.1 Machines

The Coda architecture recognizes three types of machines, *clients*, *servers* and a *system control machine* (or SCM). Client machines are typically single-user workstations used to access shared information. These machines are not trusted by servers except for the duration of a login session by an authenticated user. Server machines are secure, trusted machines whose purpose is to service client requests for shared data. As the custodians of shared information, servers must require authentication of each user before releasing shared data to the users client workstation. The third machine type is the system control machine (SCM). The purpose of the SCM is to provide a single point of control for ease of administration. Logically, the SCM is distinct from the servers, but, physically, the SCM can also act as a server.

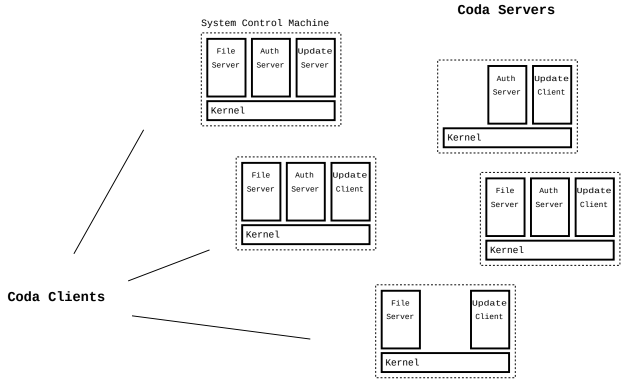

Figure 5.1: Typical Coda server organization

File Server Process	The *codasrv* process interacts withthe venus process on clients. Together they fulfill user requ
Auth Server Process	The *auth2* process runs on allservers. It validates user passwords and issues a token for that u
Update Client Process	The *updateclnt* process works in conjuctionwith the *updatesrv* process (running on the SCM)

Table 5.1: Server Processes

5.2 Processes

Each of the dedicated file servers must run a number of processes. These processes are shown in Figure 5.1 (XXX) and are described below.

Figure 5.1 (XXX) lists the typical processes of a running file server.

```
  PID TT STAT  TIME COMMAND
    0 ?  R <   53hr  (kernel idle)
    1 ?  SW    0:00     init -a (init)
    2 ?  U <  10:00  (inode_pager)
    3 ?  S <   0:00  (device pager)
    4 ?  S <   0:00  (device server)
    5 ?  S     0:00  (exception hdlr)
    6 ?  SW    0:00 /etc/mach_init -a
  152 ?  SW    0:00 updatesrv       -s -p /vice/db
  156 ?  SW    0:00 auth2
  179 ?  SW    3:19 /etc/update
  182 ?  SW    0:00 /etc/cron
  190 ?  SW    0:07 /usr/cs/etc/nanny -init /usr/cs/etc/nanny.config
  202 ?  SW    0:00 /etc/inetd
  203 ?  SW    0:07 /etc/named
  204 ?  SW    0:00 /usr/cs/etc/supfilesrv
  205 ?  SW    0:00 /usr/cs/lib/lpd -l
  206 ?  SW    0:00 RFS Listener.
 1217 ?  SW    0:00 sh /vice/bin/startserver
 1222 ?  S     6:56 codasrv           -time -maxworktime 15 -nodumpvm -trunc 1 -rvm
/dev/rrz3c /dev/rrz0h 69206016
 1358 t0 R     0:00 ps axww
```

5.3 Data Location

The information stored on Coda servers is organized into several directories. These directories are are described below.

- /**vice**/**auth2** This directory contains information related to the authentication process, including its log file.
- /**vice**/**bin** contains the Coda file system binaries for servers and the SCM.
- /**vice**/**db**contains the log file for the update processes as well as a number of databases important to servers.

- /**vice**/**srv**contains information related to the server process, including its log file.
- /**vice**/**vol** contains information related to the volumes contained in the Coda file system.
- /**vice**/**vol**/**remote** exists only on the SCM and contains volume information for all remote servers.

A more detailed description of the files in the these directories is listed in Appendix B (XXX).

5.4 File System Consistency

After a crash, many subsystems of Coda must be checked to maintain file system consistency. To ensure that Coda data partitions are internally consistent, CMUs modified version of fsck, is run during reboot. Coda accesses files by inode rather than name. The normal fsck would toss out Coda files because they have no name. The *salvager*, run by the file process during initialization, checks that every inode is referenced by at least one Coda directory. Thus, CMUs fsck and salvage together provide what fsck normally provides for the Unix file system. The reason fsck is not a sufficient check on Coda file systems is that it does not recognize Coda directories (Coda directories appear to be regular files in the Unix file system) so it cannot perform the inode reference check.

Chapter 6

Configuring Kernels for use with Coda

Currently, Coda clients and servers are supported under Linux, FreeBSD and NetBSD.

The Coda servers may be run on unmodified kernels. The Coda `codasrv` process is user-level code that uses existing kernel services. The Venus client, however, requires some Coda-specific changes be made to the kernel. The kernel specific changes add Coda specific definitions to the Vnode layer in the kernel which are needed to satisfy the requirements of the Coda file system to the vfs layer in the kernel.

6.1 VFS Interface

In Unix systems, the code which translates user-generated system calls into file system requests is called the VFS Interface. Coda provides a VFS Interface by providing the necessary support for systems calls such as creat(), open(), stat(), etc.

Communication between Venus and the kernel occurs through a character device `/dev/cfs0` (part of Coda) which provides the kernel with access to Venus for handling the vsf<->vnode kernel interface. Please see 8 (Installing and Configuring a Coda Client) for the major and minor device numbers specific to a supported platform.

The rest of this chapter will focus on integrating the Coda code that satisfies the Vnode interface into the kernels of supported platforms.

6.2 Configuring a Linux kernel

Coda may be configured as a module or hard-coded in a "monolithic" kernel. We have a reasonably flexible method to build kernel modules. You can build the module for a kernel which you are not running at the time of the build.

To build a custom coda kernel module you need to obtain the tarball from:

ftp://ftp.coda.cs.cmu.edu/pub/coda/linux/src/linux-coda-4.6.2.tgz

And follow the instructions in the rest of this file.

NOTE: The code here is always newer than or equal to:

- the kernel
- the big Coda tarball

6.2.1 To Build 2.1 kernels modules

- Patch the kernel:

  For 2.1.10? kernels apply the 106patch. This will go to Linus shortly, so this patch may stop working.

  You must type the following:

  ```
  make oldconfig
  make dep
  make your-type-of-image and modules.
  ```

- In this directory build the coda.o module:

  type:

  ```
  make config  --- answer the questions
  make coda.o
  su
  make install
  ```

6.2.2 To Build 2.0 kernel modules

- Prep the kernel by typing:

  ```
  cd /usr/src/linux-for-you
  make oldconfig
  make dep
  ```

- In this directory build the coda.o module by typing:

  ```
  make config  --- answer the questions
  make coda.o
  su
  make install
  ```

6.2.3 NOTES

- If you build for a running kernel, you must still have a source tree for that Linux release, otherwise the headers cannot be found. However, that source tree doesn't need modversions.h etc.
- We don't actively maintain 2.0 code anymore. The 2.1 code is preferred, but not yet perfect.

6.3 Configuring a FreeBSD kernel

Currently, Coda works with the 2.2.x-STABLE kernels. Since a loadable module for Coda is not available, Venus support is compiled into the kernel. The Coda Development Group releases a pre-built GENERIC

FreeBSD kernel with Coda Venus support. The only differences between the GENERIC FreeBSD kernel and the GENERIC Coda kernel is the addition of Coda Filesystem and the enabling of the kernel debugger.

For those who wish to compile their own kernel, go to:

ftp://ftp.coda.cs.cmu.edu/pub/coda/freebsd/<OS-VERSION>/i386/">

and get the Coda kernel patches:

`kernel-patch.cfs-4.4.1.gz`.

If you are patching a release earlier than 2.2.6, you will also need:

`kernel-patch.FreeBSD-4.4.0-2.2.5.gz`.

You should obtain the FreeBSD 2.x kernel sources and apply the `kernel-patch.cfs-4.4.1.gz` kernel patch. Suppose that the kernel source is in `/home/me/mysrc/sys` and the patch files, `kernel-patch.cfs-4.4.1.gz` and `kernel-patch.FreeBSD-4.4.0-2.2.5.gz`, are in `/home/me/mysrc`. Go to `/home/me/mysrc` and extract the Coda patch by typing:

```
zcat kernel-patch.cfs-4.4.1.gz | patch -p4
```

The -p4 is really necessary. If you are patching FreeBSD Release 2.2.5 or earlier, you also need to extract the "glue" code for Coda. Type:

```
zcat kernel-patch.FreeBSD-4.4.0-2.2.5.gz | patch -p6
```

At this point, the easiest thing to do is copy the GENERIC kernel configuration and modify it to include Coda:

```
cd .../conf
cp GENERIC Coda
```

The two lines that need to be added to the Coda configuration will look something like:

```
options             CFS     #Coda File System
pseudo-device vcfs  4       # coda minicache <-> venus comm.
```

Look at the LINT file in the configuration directory to find the most current version of these lines and copy them into the Coda configuration. You should also copy the line from LINT that enables the ddb debugger into the Coda configuration.

Now, all that you need to do is to `config` Coda and build it.

```
config Coda
make
```

Finally, you should install this new kernel by becoming root and typing:

```
make install
```

6.3.1 Epilog

If you are looking for the above Generic Coda kernel, you can find it in the FreeBSD area on the Coda site. It is named `kernel.gz`

If you do not want to apply the Coda patch (above) and you are at FreeBSD release 2.2.6 or better, you can simply copy the `cfs/` directory from the Coda source tree to `/home/me/mysrc/sys`

```
cd <coda sources-coda_release>/kernel-src/vfs/bsd44/
cp -rp cfs /home/me/mysrc/sys
```

Don't forget to check `INSTALL` and `README` in the Coda FreeBSD ftp area for any last minute changes that may not have found their way into the Coda manual. Also, remember you only need a modified kernel for the client side of Coda, not the server side.

6.4 Configuring a NetBSD kernel

Currently, Coda only works with the 1.3/1.3.x kernels. Since a loadable module for Coda is not available, Venus support is compiled into the kernel. The Coda Development Group releases a pre-built GENERIC NetBSD kernel with Coda Venus support. The only difference between the GENERIC NetBSD kernel and the Coda kernel is the addition of Coda Filesystem.

For those who wish to compile their own kernel, go to:

ftp://ftp.coda.cs.cmu.edu/pub/coda/NetBSD/<OS-VERSION>/i386/">

and get the Coda kernel patches:

```
kernel-patch.cfs-4.4.0.gz
kernel-patch.NetBSD-4.4.0-<OS-VERSION>.gz
```

You should obtain the NetBSD 1.3/1.32 kernel sources and apply the coda patches. Suppose that the kernel source are in `/home/me/mysrc/sys` and the patch files, `kernel-patch.cfs-4.4.0.gz` and `kernel-patch.NetBSD-4.4.0-<OS-VERSION>.gz`, are in `/home/me/mysrc`. Go to `/home/me/mysrc` and extract the Coda patch by typing:

```
        zcat kernel-patch.cfs-4.4.0.gz | patch -p4
```

The -p4 is really necessary. You also need to extract the "glue" code for Coda. Type:

```
zcat kernel-patch.NetBSD-4.4.0-<OS-VERSION>.gz | patch -p6
```

At this point, the easiest thing to do is to copy the GENERIC kernel configuration and modify it to include Coda, i.e.

```
cd .../conf
cp GENERIC Coda
```

The two lines that are need for to the Coda configuration will look something like:

```
options             CFS     #Coda File System
pseudo-device vcfs  4       # coda minicache <-> venus comm.
```

These should have been added to the GENERIC file by the above patches. Please verify that they went in correctly.

Now, all that you need to do is to `config` Coda and build it.

```
config Coda
make
```

Finally, copy the `netbsd` to / and reboot.

6.4.1 Epilog

If you are looking for the above Generic Coda kernel, you can find it in the NetBSD area on the Coda site. It is named `netbsd-<RELEASE>.gz`

Don't forget to check `INSTALL` and `README` in the Coda NetBSD ftp site for any last minute changes that may not have found their way into the Coda manual. Also, remember you only need a modified kernel for the client side of Coda, not the server side.

Chapter 7

Installing a Coda Server

7.1 Introduction

A Coda cell is a set of servers which all believe one member of the set to be the master, or SCM server. All modifications to important Coda databases should be made on the SCM, otherwise the SCM plays the role of an ordinary Coda server. The `updateclt`/`updatesrv` daemons will then propagate changes from the SCM to the other servers.

The first server setup must be the SCM. This chapter is divided into three sections: installing the binaries, configuring the system using the configuration script `vice-setup` (for both SCM and non-SCM servers) and finally, a detailed look at what needs to be done to setup a server manually. Before beginning, we have provided some overview information to assist in setting up the server.

7.1.1 Recoverable Virtual Memory

To help ensure that data is not lost or left in inconsistent state between server restarts, Coda uses *Recoverable Virtual Memory* (RVM) which is a transaction system that maintains the system state of the Coda server meta data. RVM is a data logging transaction system which writes modifications into the RVM log and upon truncation or restart incorporates such modifications into the RVM data file. Both log and data file for RVM are ideally held on raw partitions.

NOTE: this should not be confused with Virtual Memory.

Upon startup, the Coda servers use RVM to restore the Coda system state. To efficiently use this feature, you should have dedicated disk partitions for optimal performance the log partition ideally on its own disk. However, a disk sharing the log partition with other disk partitions or the use of a log file may be used at corresponding loss of optimal performance. See section 7.4.1 (XXX) for more details on RVM.

7.1.2 Server Disk Organization

Coda servers require a minimum of 2 disk partitions for optimal performance (one raw partition for the RVM Log, one raw partition for RVM Data and one regular UNIX filesystem to hold file data store in Coda), data security and protection from accidental deletion. For additional performance gains the RVM Log partition

Partition	Storage Purpose	Mounted	Typical Size	Whether fscked
hda2	RootandUser File System	/	650MB	Yes
hda5	Var file system	/var	100MB	Yes
hda3	Vice File System	/vice	300MB	Yes
hdc1	RVM Log	Not	12MB	No
sda1	RVM Data	Not	130MB	No
sda2	Coda FS Data0	/vicepa	1.6GB	Yes
sda3	Coda FS Data1	/vicepb	1.6GB	Yes
sda5	Coda FS Data2	/vicepc	1.6GB	Yes

Table 7.1: Example of Partitions for a Coda Server

should be on its own disk to avoid head movement latencies which reduces disk seek times on log operations. Optionally, `/vice` can be a separate partition for the same reasons it is advantages to have `/var` as a separate partition.

However, other configuration may be used such as having the RVM Data and Log information stored as regular UNIX files at a loss in performance and data security. Also, if more than one Storage Area Data is needed on a Coda Server (the default directory is called `/vicepa`), the additional storage areas must be separate partitions (different partition from `/vicepa` the default, initial storage area for data under Coda) and mounted as `/vicepb`, for example.

The table below shows a possible partitioning of disks on Coda servers with their respective purpose, mount points, typical sizes and consistency check program. Please note that the sizes of these partitions were taken from one of the Coda servers at CMU-SCS: the actual sizes may vary at other installations.

7.2 Obtaining Coda

Currently, the binary and source packages are available via ftp from

(*ftp://ftp.coda.cs.cmu.edu/pub/coda/*<platform>)

as well as the Coda Project's local Coda servers (`/coda/project/release/objs/`<platform>) for CMU users. The platform layout for finding the Coda binaries relative to the base directory indicated above is:

```
linux/<coda-release>/{i386,sparc,coda-fs-module,SRPMS} # For Linux
freebsd/<coda-release>/<OS_RELEASE_VERSION>/i386          # For FreeBSD
netbsd/<coda-release>/<OS_RELEASE_VERSION>/i386        # For NetBSD
```

Where <coda_release> refers to the Coda version and <OS_RELEASE_VERSION> is the version of the Operating System. For example, if you want Coda release 4.3.13 for i386 FreeBSD 2.2.5, the correct platform directory is: `fbsd/4.3.13/2.2.5/i386/`. For i386 Linux, it is: `linux/4.3.13/i386/`, and for i386 NetBSD 1.3, it is: `netbsd/4.3.13/1.3/i386/`. For more information on the layout, please check the document `LAYOUT` available in the top-level of the locations listed above for obtaining Coda. However, the following files are needed by the various platforms:

Linux

```
coda-debug-server-<release>.i386.rpm
coda-doc-<release>.i386.rpm
coda-debug-backup-<release>.i386.rpm
```

Note that the Coda Project supports the glibc (GNU C Library) version of Red Hat (5.X and higher). Support for the older Linux standard Lib C is no longer provided although Coda is known to run on many other Linux systems.

FreeBSD and NetBSD

```
coda-server-<release>.tgz
coda-doc-<release>.tgz
```

7.3 Installing and Configuring A Coda Server

Currently, Linux, NetBSD and FreeBSD are supported (CMU-MACH is no longer supported). For Linux, the Red Hat Package Management tool is supported. The package system of FreeBSD 2.X and higher as well as NetBSD 1.3+ is now supported.

TIP: Always check the INSTALL.<platform> on (*ftp://ftp.coda.cs.cmu.edu/pub/coda/*<platform>) for last minute changes and updates that have not yet found their way into the manual.

7.3.1 Installing the Coda Server Binaries and Documentation

Currently, server excutables for the supported platforms are installed into the following directories:

Linux

installs files/binaries in `/etc`, `/usr/bin` and `/usr/sbin`.

FreeBSD

installs files/binaries in `/usr/local/etc`, `/usr/local/bin` and `/usr/local/sbin`

NetBSD

installs files/binaries in `/usr/pkg/etc`, `/usr/pkg/bin` and `/usr/pkg/sbin`

*NOTE:*Please make sure your PATH environment variable is set properly for your platform. That is, ensure FreeBSD has `/usr/local/{bin,sbin}` and NetBSD has `/usr/pkg/{bin,sbin}` in its default search paths. Linux installs Coda into the default system paths, so Linux shouldn't need its default seach path modified.

There is one directory used by all platfroms: `/vice` which is used to store the supporting Coda configuration and Coda databases. It does not need to be in the PATH.

Specific steps to install the binaries and documentation are:

Linux

```
rpm -Uvh coda-debug-server-<release>.i386.rpm
rpm -Uvh coda-doc-<release>.i386.rpm
```

NetBSD and FreeBSD

The installation procedure is identical for both platforms (we assume you are using GNU tar which is the default tar for both platforms):

```
pkg_add coda-server-<OS_RELEASE_VERSION>-<coda_release>.tgz
tar zxfv coda-doc-<release>.tgz -C /usr/share
```

Please note that NetBSD uses a different location for NetBSD packages (/usr/pkg instead of /usr/local). Please make sure /usr/pkg/{bin,sbin} are in your paths under NetBSD.

7.3.2 Configuring A Coda Server

Server setup is similar for all platforms.

Before Beginning

The `/vice` directory will be created by the setup script automatically if it does not already exist. However, if your root partition does not have enough additional space (e.g. the SrvLog file can become huge +10MB under certain circumstances), you will need to create a partition to be mount as `/vice`, format the partition as normal UNIX file system, create the mount point `/vice`, add the partition and mount information to `/etc/fstab` and mount it by hand *BEFORE* running the setup script.

Alternatively, `/vice` may be a symbolic link to a directory on an existing partition with enough space but it also must be created before running the setup script, `vice-setup`. Sub-directories needed under `/vice` will be created by `vice-setup`.

Also note that the setup of an SCM differs in some important ways from a non-SCM setup. So, when `vice-setup` asks: *Will this machine will be the SCM?* it is very important this question is answered correctly.

Only one SCM should be setup per Coda Cell and Coda does not support multiple-homing cells at this time.

While `vice-setup` sets up things that are common for both the SCM and non-SCM servers, the setup of scripts called by answering yes to the *Is this the SCM?* are:

SCM Setup

`vice-setup`

```
vice-setup-scm
vice-setup-user
vice-setup-rvm
vice-setup-srvdir
```

The set of scripts called by answering no are:

non-SCM Setup

`vice-setup`

```
vice-setup-rvm
vice-setup-srvdir
```

in the order listed above.

Setting Up A Coda Server

If this is the first (or only) server being setup for a Coda cell, it must be setup as the "SCM" by answering yes to the question when asked by `vice-setup`. The SCM coordinates the sync'ing of Coda database and global configuration files needed to keep track of data stored in Coda. The SCM also coordinates the authentication of a Coda user.

If you are adding a non-SCM server which keeps copies of databases, but does not distribute them to the rest of the machines comprising a Coda Cell answer no to the question.

Other than the distinctions indicated above, the SCM plays no special role in a Coda cell and can actually be down for a short time without denying service in a multiple-machine cell site. However, is extremely important to only have one SCM

NOTE: that running a server seriously dips into system virtual memory. Running a Coda server, a Coda client *and* X11, the Coda Group as observed that a slightly over 64MB of available VM is needed. The command `top` gives information on memory, cpu and process usage. Therefore, we recommend not running a Coda client on a Coda server and only run X when performing server maintenance if X is convenient to use.

Whether setting up an SCM Coda server or adding a server to an existing Coda Cell, make sure the following items are taken care before running the setup script:

1. an empty directory (/vicepa) where the file server will put user files. If more than one storage are are is needed for file space on an existing server, such as needed `/vicepb`, additional space must be a separate partition from `/vicepa`.
2. a raw partition for RVM metadata (you can use a file but this will be slow on a medium to large server) This partition must be around 4% of the total size of the files you wish to store under /vicepa (e.g. on a 3.3GB server we use around 130M of rvm data). Consider 10M to be the minimum.
3. a LOG partition, preferably on a disk by itself. This needs not be large (12MB is large enough for a 3.3GIG partition which is the largest default configuration provided by `vice-setup-rvm` script).
4. two secret tokens of _exactly_ 8 characters (eg elephant). The RVM files involve the journalling/transactional aspects of Coda and communicating this information between servers and via the loop-back device when the SCM updates itself. The tokens are used to secure intra-server communication.
5. If the root partition is not large enough to hold `/vice`, having alternative measures setup as outlined in 7.3.2 (Before Begining).

Once you have the above items available, run:

```
vice-setup
```

This provides several useful "canned" configuration for setting up a Coda server. It does a number of things "behind the scenes" as well (such as setting up the directory `/vice` if it doesn't exist and populating it with the sub-directories `backup db srv vol bin`) while asking the following questions:

Enter a random token for auth2 authentication :

This must be _exactly_ eight characters in length. This requirement is due to a bug that hasn't been completely fix yet.

Enter a random token for volutil authentication :

This must be _exactly_ eight characters in length and different that the auth2 token. This requirement is due to a bug that has not been completely fixed yet.

Do you want to start the server at boot time? (y/n)

Answering yes to this question creates the file `/vice/srv/STARTFROMBOOT` which must be present for either `rc.vice` or `vice.init` to start the server at boot time. Removal of this file will prevent the Coda server from starting automatically at boot time.

Is this the master server, aka the SCM machine? (y/n)

For the first Coda machine being setup in a Coda cell, the answer must be "yes". If you are adding a server to an existing Coda cell, the answer must be "no".

If you have answered “no” to the above question, you will be asked for the HOSTNAME of an existing SCM:

If you have answered “yes” to the above question, you will see the message:

```
An initial administrative user "admin" with password "changeme" now exists
```

. and pick up at the question *Are you ready to setup up RVM?*

If you have answered “no” to the above question, you will be asked for the HOSTNAME of an existing SCM:

Enter the hostname of the SCM machine :

This question is only asked if the machine is not the SCM. The hostname entered must be that of the SCM which distributes the Coda cells’database and global configuration files.

Are you ready to set up RVM? [yes/no]

Answer yes to this question or the setup will abort.

What is your log partition?

RVM is the Recoverable Virtual Memory used to keep a copy of Coda Virtual memory on local disk. While this can be a file, it is strongly recommended that it be a small partition on a dedicated disk. If you have set aside, for example, /dev/sdc1 under Linux as the log partition, enter the partitions’ device name at the prompt. If you are setting up a *BSD system, you must enter the raw partition name. For example, if you are going to use `/dev/sd2e` you must enter the raw device name `/dev/rsd2e`. The maximum template size we provide for and have tested is 30MB.

What is your log size? (enter as e.g. ’12M’)

The log partition keeps transaction records for Coda volume updates that may not have been writen to the Coda data partition yet. We have not extensively tested values above 30MB and don’t recommend more than this. 12M is a good value for a small to medium sized server. The value may be entered as, 12M or 12288 for 12 mega-bytes. `Please see section 7.4.1 (The Data Partition) for a more detailed explanation`

What is your data partition (or file)?

This allows you to specify a partition or file. We strongly recommend a partition which can reside with other system partitions. Remember, if you use a partition, you are using it "raw" . it does not contain a file system of any kind. Again, if you are using Linux, you may enter a block device name such as `/dev/sdc1`. If you are using a *BSD system, you must enter a raw device such as `/dev/rsd2e`.

What is the size of you data partition (or file) [22M,44M, 90M, 130M : "]

This specifies the the size of the RVM data partition. It must be typed exactly as `22M`, `44M` , `90M` , or `130M` . These are associated with default parameters that must be feed to `codasrv` when it is started. The following table provides the log size to total data storage conversion:

- `22M` is good for up to 500MB
- `44M` is good for up to 1.0GIG
- `90M` is good for up to 2.2GIG
- `130M` is good for up to 3.3GIG

For a detailed explanation of the trade off between different Data log sizes vs. Data Storage Space, please see the section 7.4.1 (The Log Partition).

Proceed, and wipe out old data? [y/N]

WARNING

if you have entered erronous entries for RVM log and RVM data partitions, you could damage your system. Be sure you entered correct information!

This is the last chance before any action is taken by the setup script. Enter `y` to commit or `N` to abort.

Where shall we store your data [/vicepa ?]

Older development versions of coda require `/vicepa` to be present. This version does not. You give a name other than the default we suggest. However, we strongly advise not calling it `/code` as that can be confused with venus.

Once you successfully complete `vice-setup`:

And an SCM server is being setup

you are ready to start the update server, the update client and the auth server, as well as the fileserver by typing:

For Linux:

```
/etc/rc.d/init.d/update.init start
/etc/rc.d/init.d/auth2.init start
/etc/rc.d/init.d/codasrv.init start
```

For FreeBSD:

```
/usr/local/sbin/auth2 &
/usr/local/sbin/updateclnt -h ‘cat /vice/db/scm‘  -q coda_udpsrv &
/usr/local/sbin/updatesrv -p /vice/db &
```

For NetBSD:

```
/usr/pkg/sbin/auth2 &
/usr/local/sbin/updateclnt -h ‘cat /vice/db/scm‘  -q coda_udpsrv &
/usr/local/sbin/updatesrv -p /vice/db &
```

One final step is needed to make the SCM server function. Use the root volume you specified in vice-setup in place of your-root-volume. This step creates the volume, which will be mountable by the client software.

```
createvol_rep your-root-volume E0000100 /vicepa
```

Once you have done this, check that the server has started by viewing the log:

```
xterm -e tail -f  /vice/srv/SrvLog &
```

If this is a non-SCM server

You should start auth2, updatesrv and updateclnt first by typing:

For Linux:

```
/etc/rc.d/init.d/update.init start
/etc/rc.d/init.d/auth2.init start
```

For FreeBSD:

```
/usr/local/sbin/auth2 -chk &
/usr/local/sbin/updateclnt -h ‘cat /vice/db/scm‘  -q coda_udpsrv &
/usr/local/sbin/updatesrv -p /vice/db &
```

For NetBSD:

```
/usr/pkg/sbin/auth2 -chk &
/usr/pkg/sbin/updateclnt -h ‘cat /vice/db/scm‘  -q coda_udpsrv &
/usr/pkg/sbin/updatesrv -p /vice/db &
```

This will cause the new server to first obtain the current server files such as `/vice/db/ROOTVOLUME` and `/vice/db/VRList` as well as others needed for this server to participate in the Coda cell. Once these files have been retrieved, you may start the server manually for the first type by typing:

For Linux:

```
/usr/sbin/startserver &
```

For FreeBSD:

```
/usr/local/sbin/startserver &
```

For NetBSD:

```
/usr/pkg/sbin/startserver &
```

Once you have done this, check that the server has started by viewing the log:

```
xterm -e tail -f  /vice/srv/SrvLog &
```

NOTE: the above distinction between SCM and non-SCM server is non-trival.

If the `codasrv` process started correctly, you should see “File Server started”, if not you need look at the messages in the SrvLog to determine what went wrong. If the messages are not helpful, you should kill the server (*NOTE:*The server is designed to continue to stay up so gdb can be attached for debugging):

```
kill -9 ‘cat /vice/srv/pid‘
```

and then you can restart the server with:

```
startserver -d 10 &
```

7.3.3 Basic Trouble Shooting and Misc.

However, assuming that the server came up, also make sure updatesrv and updateclnt are running by using either `top` or `ps`.

NOTE:

> you need the package bc (binary calculator on the server). you can either obtain from our ftp site or a mirror site containing your operating system and contrib software.

NOTE2:

> E0000100 is the Volume Storage Group set up for you by vice-setup when setting up an SCM. With more servers you can define other groups in /vice/db/VSGDB.

Once this is complete, you should setup a client machine and run Venus and point `venus` point to this server. Setting up a client machine with `venus` is covered in the next chapter. Once you have tested and verified the Coda server works from a client, you are ready to adduser more volumes and users!

NOTE: A current limitation of the current vice-setup-user script is that the initial admin account needed is assigned a Coda uid of 500. In order for a client to authenticate a user, that user must exist with the same corresponding user-id in the UNIX password file. If you need to change the admin account uid to suite local uid space, please edit the script `vice-setup-user`, change the default Coda uid to the needed value and re-run the script. This only applies if you are setting up the SCM.

Otherwise, if this is a non-SCM server, you might consider setting up a singlely replicated volume on the new server to test it.

7.4 Underneath vice-setup

The sections contained here describe what `vice-setup` does for you. This information is useful for those who either wish to customize our default vice-setup script or wish to have a custom server setup outside the scope of Coda's setup script..

7.4.1 RVM Initialization

RVM initialization requires the selection of several parameters, each of which involve tradeoffs. Although the RVM log and data can be kept as regular UFS files, **this is not recommended**: Raw partitions have much stronger data consistency guarantees as well performance advantages. It is probably best to plan out the RVM partitions on paper first, taking into consideration both the effect on RVM performance as well as overall disk usage if you choose not to use the values provided in the `vice-setup` script.

The Log Partition

The size of the log device is based on available space and issues involving truncation. Larger logs provide a longer accessible history of operations, are truncated less frequently, but each truncation will take a longer period of time. Shorter logs truncate more often, but each truncation takes less time. Log size is also strongly related to server startup time as well. We use a 90M log size, on a storage size roughly 3.2GIG spread between two UNIX partitions. (We suggest leaving a little space at the end of the RVM log partition for safety, as RVM automatically adds about one extra page to the amount you specify). Our `vice-setup` script provides for several default values, the largest of which is 130MB. If you decide to follow the recommendation and use a dedicated partition, creating a 130MB partition will leave plenty of room to expand even if your initial use only suggests a size of 22M, for example..

The log is initialized with **rvmutl**. At **rvmutl**s prompt, use the command **i**, and then specify the size of the log segment. In specifying the size, you can use M for megabyte and K for kilobyte. For example, to initialize a log on partition 0g to eight megabytes:

```
# @B(rvmutl)
* i
Enter name of log file or device: /dev/sdc1
Enter length of log data area: 22M
* q
```

The Data Partition

The data segment contains the meta-data of the system such as volume headers, Coda directories, vmond lists, resolution logs, etc. The size of the data segment depends on the amount of disk space for file data, i.e. the size of the /vicep? partitions. As a rule of thumb, you will need about 3-5% of the total file data space for recoverable storage. We currently use 4% as a good value under most circumstances. In our systems the data segment is 90Meg for approximately 3.2 gigabytes of disk space. By making it smaller, you can reduce server startup time. However, if you run out of space on the RVM Data partition, you will be forced to reinitialize the system, a costly penalty. So, plan accordingly.

In initializing (or reinitializing), you need to pick several parameters. The first is the starting address of the recoverable segment in your address space. On our servers we start the RVM segment at 0x20000000 for both Linux and BSD based systems on Intel architecture. The second is the amount of space to give the recoverable heap. The heap will obviously grow over use, so plan accordingly. Our heap space is 0x1000000. We suggest that for the last parameters you use 1Meg (0x100000) for the static area, use 80 free lists (or nlists), and a chunksize of 32. These numbers will work well with the internal structure of the fileserver and are provided as the default vales in the setup script `vice-setup`.

To perform the data initialization, run the program **rdsinit**. **rdsinit** takes two parameters, the names of the RVM log and data devices. For example, to initialize one of our Intel based servers:

```
# rdsinit /dev/hdc1 /dev/sdb1
Enter the length of the device /dev/hdc1:119070700
Going to initialize data file to zero, could take awhile.
done.
rvm_initialize succeeded.
starting address of rvm: 0x20000000
heap len: 0x1000000
static len: 0x100000
nlists: 80
chunksize: 32
rds_zap_heap completed successfully.
rvm_terminate succeeded.
```

NOTE: the use of the decimal value for the length of the device and the use of hex values for the address and lengths of the next three values.

7.4.2 Update Monitor

The update monitor is used to propagate changes to the Coda server databases to all servers from the SCM. Client update processes run on all the Coda servers and connect to a server udpate process running on the SCM. The server process uses the file `/vice/db/files` to determine which files should be kept consistent on all the servers. See the *updateclnt*(8) and *updatesrv*(8) man pages for more details.

Create the file `/vice/db/files` on the SCM. Currently our `/vice/db/files` looks like this:

```
VLDB
auth2.pw
auth2.tk
pro.db
servers
hosts
vice.pdb
vice.pcf
volutil.tk
VRDB
files
VSGDB
dumplist
scm
ROOTVOLUME
```

7.4.3 Authentication Database

Coda uses an authentication database that is separate from the UNIX password file. This database is maintained by the SCM. When someone authenticates to Coda, their password is checked against this database and that person is issued a token if they successfully authenticate. This section describes how to initialize the authentication database.

On the SCM

If the server you are installing is the SCM, you must set up the initial authentication database. Make sure you add at least one user to the `System:Administrators` group. You need to do the following:

- Create or modify an appropriate user file in the Vice format **user_coda (5)**. An example of `user.coda` can be found in the directory `/vice/db` as well as in Appendix D (XXX).
- Create or modify an appropriate group.coda file in the Vice format **group.coda (5)**. It is vital that you add at least one user from the `user.coda` file to the group `System:Administrators`. An example of `group.coda` can be found in the directory `/vice/db` as well as in Appendix D (XXX).
- `/usr/sbin/pwd2pdb -u user.coda -g group.coda > vice.pdb`

- `/vice/bin/pcfgen vice.pdb` which will create vice.pcf.
- Create a file /vice/db/passwd.coda, owned by root, readable only by root, with the initial passwords for users in the Vice passwd file format (passwd_coda (5)). The format of the passwd.coda file should be as follows:

  `uid<TAB>actual cleartext password<TAB>any desired info`.

  The information field could be the full name of the user. Name this file `/vice/db/passwd.coda`. Then run the command

  `initpw -k "drseuss " </vice/db/passwd.coda > /vice/db/auth2.pw`

  This command produces the file `auth2.pw` in the format:

  `uid<TAB>encrypted password<TAB>Desired info`.

  Note: "drseuss " is an encryption key to prevent accidental disclosure of the passwords to the system administrators. In future versions of coda we will store a sites specific encryption key in a configuration file.

  Note that all future changes of the password must be made through the au(1) program, and not by repeating this procedure since that would overwrite any passwords changed by users after initialization. New passwords must be *exactly* 8 characters long. To add additional users, please see Section 11.1 (XXX).
- You also must choose a password for the servers to use in generating the *secret* token (see Chapter 5 ()) for more information. This secret token is used for secure communication. Put the password in `/vice/db/auth2.tk`.
- Create the ticket file `/vice/db/volutil.tk`, used for secure communication among the servers. This file should contain a single word which will be used to encrypt messages between the server and server applications. The file should be present on all servers. The UpdateMon presently takes care of propagating them from the SCM.

On Servers OTHER than the SCM

On servers other than the SCM, the **Update** monitor will ensure that the proper database files are propogated from the SCM. However, you must copy the file `/vice/db/volutil.tk` to the new server for the **Update** monitor to work.

7.4.4 Creating the Volume Storage Group DataBase (SCM only)

Every combination of servers setup expects to use as a VSG which must be given a multicast address in `/vice/db/VSGDB`. This file must be created by hand; please refer to **VSGDB (5)** in Appendix E (XXX) for more details.

7.5 Starting the File Server

We have provided a script called `startserver` which reads in the file `/vice/srv.conf` which contains the startup information needed by `codasrv`. The syntax of `/vice/srv.conf` is:

```
-rvm <log_partition> <data_partition> <size_of_log_partition>
```

The `vice-setup-rvm` contains the canned default value for the corresponding value entered through `vice-setup`. Should you deviate from one of these canned default values, you will need to either create or modify the `/vice/srv.conf` file to indicate the correct data size of the data partition. For example, one of CMU's Coda servers contains:

```
-rvm /dev/hdc1 /dev/sda1 119070700
```

You should be careful to specify the correct partitions for the RVM log, data segment, and size of the data segment.

Once this file has been correctly edited, you can start the server with **startserver&**.

7.5.1 Creating the Root Volume

If you are installing any server other than the SCM, please use the instructions in section 10.2 (XXX) to create new volumes on this server.

The root of the Coda file system must reside on the /vicepa partition for correct functioning. Since the SCM probably also acts as a server, place the root of the Coda file system in the /vicepa partition on the SCM. Create the root volume by using either one of the following commands

```
% createvol_rep codaroot <VSG_ENTRY> /vicepa
```

```
% createvol codaroot <hostname> /vicepa
```

the former is for replicated root volume, where <VSG_ENTRY> is the entry in the `VSGDB` for the volume storage group for this volume; the later is for non-replicated root volume, where <hostname> is the name of the host which hosts the root volume. Since it is likely that the root will be cloned and read-only replicated, you might wish to create this volume as singly replicated for simplicity.

These commands will create the `/vice/db/VLDB`, `/vice/db/VRDB`, `/vice/vol/VolumeList`, `/vice/vol/BigVolumeList`, and the `/vice/vol/AllVolume` files.

Create or modify the file `/vice/db/ROOTVOLUME` to contain the volume name (E.g. `codaroot`) corresponding to the root volume as listed in the file `/vice/vol/VRList`.

Creating a readonly root

You may wish for the root to be readonly. To do this, first create a root volume as above. Start a venus which can talk to this server. Setup the directory structure you want in your root volume. Once you are satisfied with the structure, clone the volume and dump it to a disk file with the the `volutil` program. Now restore the dump file to the servers in the desired VSG for the readonly volume. Be sure to specify the volumeId and the volume Name for the new volume:

```
% volutil restore <filename> /vicepa <newrootname> <volid>
```

Then modify the `/vice/db/ROOTVOLUME` file to contain the new rootvolume name, <newrootname>.

Chapter 8

Installing and Configuring a Coda Client

A Coda client can be run on any workstation that has kernel support for Coda in its kernel. This chapter assumes the chapter on 6 (Configuring Kernels for use with Coda) has been read. It contains valuable information for building a kernel if a pre-built kernel with Coda support (or loadable coda kernel modules) cannot be used. Supported UNIX$^{\mathrm{TM}}$-like platforms are currently Linux, FreeBSD and NetBSD.

8.1 Installing and Configuring the Coda Client Binaries and Documentation

We currently provide binary distributions of the Coda client software for Red Hat Linux (i386 and Sparc), FreeBSD and NetBSD (i386). Red Hat Linux's rpm package format is supported. NetBSD 1.3.X and FreeBSD package format are supported as well. Please consult the `INSTALL.<platform>` available on *ftp://ftp.coda.cs.cmu.edu/pub/coda/*`<platform>` for the current version of Coda and for last minute updates which possibly have not made their way into the manual. To install the binaries for each platform, the following instructions apply:

For Linux

Obtain: *ftp://ftp.coda.cs.cmu.edu/pub/coda/linux/i386/*

- `coda-debug-module-<kernel_version>-<coda_release>.i386.rpm`
- `coda-debug-clients-<kernel_version>-<coda_release>.i386.rpm`
- `coda-doc-<coda_release>.i386.rpm`

NOTE: We provide a binary glibc (GNU C library) version of Coda for Red Hat 5.X and higher. Coda has run successfully on other versions of Linux and with different C libraries too.

To install the binary and documentation packages, do:

- `rpm -Uvh coda-module-<kernel_version>-<coda_release>.i386.rpm`
- `rpm -Uvh coda-debug-clients-<kernel_version>-<coda_release>.i386.rpm`
- `rpm -Uvh coda-doc-<coda_release>.i386.rpm`

FreeBSD and NetBSD:

For FreeBSD, obtain: *ftp://ftp.coda.cs.cmu.edu/pub/coda/freebsd/*`<OS-RELEASE>/i386/`

For NetBSD, obtain: *ftp://ftp.coda.cs.cmu.edu/pub/coda/netbsd/*`<OS-RELEASE>/i386/`

Once the binary and documentation packages have been obtained, the following will install them:

- `pkg_add coda-clients-<OS_RELEASE>-<coda_release>/`
- `tar zxfc coda-doc-<coda_release>.tgz -C /usr/share`

Please make sure the correct path for your platform is set in your environmental PATH variable:

For Linux:

Coda installs in the standard /usr tree:

```
/usr/sbin
/usr/bin
```

For FreeBSD:

Coda installs in the /usr/local tree:

```
/usr/local/sbin
/usr/local/bin
```

which is not usually part of the FreeBSD path search, particularly for the root account. Please make sure this is part of your seach PATH.

NetBSD:

Please note the NetBSD has elected to create a new tree to install NetBSD packages into (NetBSD uses `/usr/pkg` instead of `/usr/local`). Coda creates package based on this `/usr/pkg` tree. Please make sure this tree is in your search path as the /usr/pkg is not part of the default PATH settings under NetBSD.

Once the binaries and documentation are installed, configuration of Venus is identical for all platforms when `venus-setup` is used. To run the script, type:

```
venus-setup <comma,seperated,list,of,servers> <cache_size_in_kilobytes>
```

. For example,

```
venus-setup micky,minnie,goofy 40000
```

will setup up `venus` to look for the set of servers micky, minnie and goofy and configure Venus to use 40MB of disk space for caching. This information is kept in `/usr/coda/etc/vstab`.

*NOTE:*At least one Coda server must be specified and the minimum cache size is 10MB (`10000`).

This script not only creates `/usr/coda` and the necessary sub-directories, but creates `/usr/coda/etc/vstab`, and adds necessary Coda port numbers to `/etc/services`.

To start venus for the first time, become root and issue the following commands:

```
# venus -init &
# xterm -e tail -f /usr/coda/etc/console &
```

The last command allows you to monitor Venus as it reports messages to `/usr/coda/etc/console`.

8.2 Upgrading from a previous Release

If you are upgrading Coda from a previous release, do not run `venus-setup` again! Simply obtain the packages for your platform as indicated above. Then, shutdown venus by becoming root and issue the following command:

```
# vutil -shutdown
```

Next, install the binary packages following the installation instructions above. Make sure you have the appropriate kernel for *BSD systems or the correct module for Linux systems. The `INSTALL.<platform>` document available from the ftp site will contain relevant notes about upgrading the kernel for *BSD systems and the module for Linux systems. INSTALL.<platform> will provide necessary advice on which kernel to use with a provided Coda release.

Also, the Chapter on 6 (Installing and Configuring a kernel for use with Coda) has information on configuring and building a kernel kernel (or loadable module) if a pre-build kernel or module won't work for the target platform.

Once the new binaries are installed and the any necessary kernel updates are completed, you need to start the new venus for the first time by becoming root and typing:

```
# venus -init &
```

The <-init> flag is used to re-initialize the `/usr/coda/venus.cache/`.

8.3 Configuration Changes made by venus-setup

This section describes the changes that must be made to various configuration files. Samples of the files are contained in Appendix D (XXX).

8.3.1 /usr/coda/etc/vstab

Create a `vstab` file in `/usr/coda/etc` to contain the Coda file system table information. The contents of this file are explained in the man page `vstab(5)`.

8.3.2 /etc/rc.*

To automatically start Venus at boot time, the script `rc.venus` is provided in `/usr/local/etc` for FreeBSD systems and `/usr/pkg/etc` for NetBSD. `venus.init` is provided for Red Hat Linux in `/etc/rc.d/init.d`.

Linux uses should create the following symbolic link if it doesn't already exist:

```
ln -sf /etc/rc.d/init.d/venus.init /etc/rc.d/
```

This will allow `init` to process the file and start Venus at boot time on Red Hat Linux systems.

*BSD systems should append the following line to `/etc/rc.local`:

For FreeBSD:

```
if [ -f /usr/local/etc/rc.venus ]; then
        . /usr/local/etc/rc.venus
fi
```

For NetBSD:

```
if [ -f /usr/pkg/etc/rc.venus ]; then
        . /usr/pkg/etc/rc.venus
fi
```

8.3.3 /etc/services

The following port numbers are added to `/etc/services` by `venus-setup`:

```
coda_opcons     1355/udp                        # Coda opcons
coda_auth       1357/udp                        # Coda auth
coda_udpsrv     1359/udp                        # Coda udpsrv
coda_filesrv    1361/udp                        # Coda filesrv
coda_venus      1363/udp                        # Coda venus
coda_backup     1407/tcp                        # coda backup service
codacon         1423/tcp        venus.cmu       # Coda Console
coda_aux1       1431/tcp                        # coda auxiliary service
coda_aux1       1431/udp                        # coda auxiliary service
coda_aux2       1433/tcp                        # coda auxiliary service
coda_aux2       1433/udp                        # coda auxiliary service
coda_aux3       1435/tcp                        # coda auxiliary service
coda_aux3       1435/udp                        # coda auxiliary service
```

8.3.4 /dev/cfs0

The following major and minor numbers are used for the various platforms:

- Character-special with Major 67 and minor 0 for Linux

  ```
  # mknod /dev/cfs0 c 67 0
  ```

- Character-special with Major 93 and minor 0 for FreeBSD

  ```
  # mknod /dev/cfs0 c 93 0
  ```

- Character-special with Major 46 and minor 0 for NetBSD 1.2

  ```
  # mknod /dev/cfs0 c 46 0
  ```

- Character-special with Major 51 and minor 0 for NetBSD 1.3

  ```
  # mknod /dev/cfs0 c 51 0
  ```

Please be sure to check the `INSTALL.<platform>` to make sure these numbers have not changed. For Linux and FreeBSD, the above numbers are officially registered. Coda on NetBSD at present does not have formally registered major device numbers so these are subject to change from release to release.

8.4 Where Data and Configuration Information is Stored

- `/usr/coda/etc` Coda configuration files critical for Venus startup.
- `/usr/coda/venus.cache` Cached copies of Coda files.
- `/usr/coda/spool` Save CMLs from aborted reintegrations.
- `/usr/coda/tmp` Temporary hoard related files.
- `/etc/services` Various port numbers used by Coda for client and server communication.

Chapter 9

Running Coda

9.1 Server

Each process running on a server maintains a log file of its activities. This file may be monitored by running **tail -f** on the particular file. For intance, an operator might want to run this command in a small window on the file server console. The table below shows the absolute pathname of the log file for each server process.

The processes append new log entries to the end of the log file. To control the growth of the log files over extended periods of operation, the files should be removed periodically. `SrvLog` may be removed by the **swaplog** option of the **volutil** command; this will move the current `SrvLog` to `SrvLog-1` and begin a new log. Swaplog also keep the last 6 log files, by removing, `SrvLog-7` and renaming the remaining log files. a swaplog is also performed during fileserver startup. The log files of the other processes should be removed only when their respective processes are not running.

9.1.1 Startup

Starting the servers srv process is relatively straightforward. You need only ensure that there is no file server currently running on the node, and give the file server information about the RVM log and data segments.

The pid of the last file process to run is contained in `/vice/srv/pid`. This is used by the **startserver**, one of our example scripts in Appendix D (XXX), to avoid accidentally starting a new server when one is already running. If the old server process is still running, and you wish to shut it down, follow the instructions in Section 9.1.2 (XXX). If `/vice/srv/pid` exists, but the server is not running, then either the machine crashed and the server process did not get a chance to remove the file, or the server has entered a *zombie* state. The server goes into a zombie state whenever it receives an unexpected signal. This allows us to debug it. The

Server Process	**Log File**
File Server	`/vice/srv/SrvLog`
Authentication	`/vice/auth2/AuthLog`
Update	`/vice/srv/UpdateLog`

Table 9.1: Server Log Files

end of the server log file will tell you if the server becomes a zombie process. See section 9.1.3 (XXX) for more details on a server becomming a zombie.

The server process requires three parameters that must be user-supplied: the device on which the RVM log resides, the device on which the RVM data segment resides, and the length of the RVM data segment. These parameters are the same ones that were set when RVM was initialized in section 7.4.1 (XXX). To start the server, type as root:

```
% /vice/bin/srv -nodumpvm -rvm <logdevice> <datadevice> <datalength>
```

Appendix D (XXX) contains two example scripts, **startserver** and **restartserver**, that simplify server startup. Be sure to change the three RVM parameters to suit your system. A common failure with these scripts is the presence of the file `/vice/srv/pid`; the scripts we have included remove this file unless the server completely crashed. If **startserver** or **restartserver** complain that a server is already running, `pid` might need to be removed.

9.1.2 Shutdown

To shutdown the file server gracefully, use the volume utility client procedure, **volutil**. Issue the command **volutil shutdown** and then monitor the log file for the server process (`/vice/srv/SrvLog`) until it reflects that the server has shutdown completely. For example, here are the last few lines before file server shutdown.

```
VShutdown: Taking volume usr.grajen(0xcc0006d1) offline...... Done
VShutdown: Taking volume synrgen.rep(0xcc00057c) offline...... Done
VShutdown: Taking volume i386.omega(0xcd0004fc) offline...... Done
VShutdown: Taking volume trace.rep.0(0xcc00057d) offline...... Done
VShutdown: Taking volume usr.mmv.rep.0(0xcc00057e) offline...... Done
VShutdown: Taking volume trace.backup(0xcc00057f) offline...... Done
VShutdown:  complete.
```

9.1.3 Monitoring and Debugging

The **volutil** program talks to the volume utility subsystem of the file server. The volume utilities not only allow you to perform administrative functions (like creating and purging volumes) but also help you to monitor the file system dynamically by setting the debugging level and "peeking" at system information.

The **volutil** program allows you to control the level of debugging information which is logged. Level 0 is the lowest level of debugging available *(i.e.) no extra information is printed* while the highest level of debugging available is 1000 *(i.e.) all debugging information is printed*. As with all other server output, debugging output will appear in `SrvLog`. Be careful to turn off debugging when its output is not in use since the log will grow rapidly and can easily fill the partition with debugging information.

As a debugging tool, we force the file program into a "zombie" state if it receives an unexpected signal. (When the file process becomes a "zombie," it logs a message to the log file which includes its Unix pid number). We then use **gdb** to attach to the zombie process. Now you may use gdb as you normally would to examine the process stack as it existed at the time of the error@Foot(If the "zombied" process is running on an IBM RT, you must first execute the command **setcontext OldContext** in order to set the context to the point of the error. To return to the current context, use the **restore** command. (Note that before quitting from gdb, you MUST return to the current context.)).

9.2 Client

9.2.1 Startup

To begin running the Coda file system on a client machine, become root on that machine and execute the command **/usr/coda/etc/venus**. Venus first scans the cache for files and directories. After the scan of the FSDB finishes, access to the "/coda" subtree is available. If the command **ls /coda** lists the file `NOTREALLYCODA`@Foot(This assumes that in setting up the client workstation, you have created the directory `/coda` and placed the file `NOTREALLYCODA` in it. Venus will mount the file system on top of this directory), Venus has either not finished initialization or has exited.

9.2.2 Shutdown

To shutdown a Coda file system client, execute the command **/usr/coda/etc/vutil -shutdown** as **root** on the client machine. This script kills the Venus running on the client machine and unmounts the Coda file system. Note that as part of the shutdown, Venus tries to unmount `/coda`. This will fail if any process is cded into `/coda` or any binaries are running from `/coda`.

9.2.3 Debugging and Monitoring

The Venus log file is found in `/usr/coda/venus.cache/venus.log`. All monitoring and debugging information is written to this file.

Once Venus is running you may run the command

```
% /usr/coda/etc/vutil -d level
```

to have Venus produce debugging output at the level requested. If you want debugging output generated by Venus during startup, you should use the *-d* option to Venus. The debugging information will be printed to Venus console file.

The **vutil** program is a utility to dynamically control and monitor the **venus** program. It has a number of options other than the debugging option described above. For more information, please refer to **vutil**s manual page in Appendix E (XXX).

9.2.4 Redirecting STDERR from Venus

If you give **venus** the parameter *-console filename*, stderr will print to the file `filename` rather than to the console. The default console file is */usr/coda/etc/console*.

Chapter 10

System Administration: Volumes

10.1 Description

Although users view the Coda file system as a hierarchy of directories and files, system administrators view the Coda file system as a hierarchy of volumes. Each volume contains a subtree of related directories and files which is a subtree of the entire file system. Volumes, then, parallel traditional Unix file systems. Like a Unix file system, a volume can be mounted. Thus, the root of a volume can be named within another volume at a mount point. The Coda file system hierarchy is built in this manner and is then mounted by each client using a conventional Unix mount point within the local file system. Since all internal mount points are invisible, the user sees only a single mount point for the entire Coda file system.

All system administration tasks are performed relative to a volume or a set of volumes. So, adding new users requires creating new volumes for their files and directories; quotas are enforced on volumes; and backups are performed on a per-volume basis. The volume abstraction greatly simplifies the administration of large systems. **NOTE: Quotas have not been implemented yet**.

The Coda file system provides four different types of volumes. The simplest of these is the *non-replicated volume*. Non-replicated volumes reside on a single server and are in the custody of the Coda file server they reside on. The Coda servers work with the venus processes on client workstations to provide a single, seamless view of the file system. However, if a custodian crashes or is otherwise inaccessible, its non-replicated volumes are inaccessible as well.

To partially solve this availability problem, Coda provides *replicated, read-only volumes*. This type of volume has exactly one read-write copy, but may have any number of read-only copies controlled by other servers. Changes to such a volume are made on the custodians read-write copy and then distributed to all servers with read-only copies. Read-only replication provides higher availability for volumes containing frequently-requested but infrequently-updated objects, like system binaries. In addition, read-only replication is used in performing backups on volumes.

Unfortunately, read-only replicas cannot provide high availability for all types of volumes, e.g. user volumes. Thus, Coda also provides *read-write, replicated volumes*. Read-write, replicated volumes are logical volumes which group together multiple read-write, non-replicated volumes. Coda provides protocols which allow read-write, replicated volumes to reside on a number of servers and to be accessed even when some servers are inaccessible. Although read-write replication provides everything read-only replication provides, its protocols are more expensive. Thus, read-only replication, rather than read-write replication, should be used

Volume Type	**Where Reads Performed**	bf/Where Writes/ **Performed**	**Conflicts Possible?**
Non-replicated	Only Custodian	Only Custodian	No
Read-only Replicated	Any Server with Replica	Only Custodian	No
Read-Write Replicated	Any VSG Member	Any VSG Member	Yes
Backup	Only Custodian	Nowhere	No

Table 10.1: Coda Volume Types

for volumes which change slowly but are accessed frequently. Table 10.1 (XXX) illustrates the differences between the volume types.

10.2 Creating a Volume

Typically, volumes consist of a single users data objects or other logically connected groups of data objects. Four factors should be used in dividing the file system tree into volumes.

1. The volume is the unit of quota enforcement and, potentially, accounting.
2. The rename command is prohibited across volume boundaries. Thus, manipulation of the tree structure at a granularity other than a whole volume (e.g. moving the mount point) or less than a volume (e.g. moving directories or files within a volume) is expensive. Moving a subtree of a volume to another volume requires copying every byte of data in the subtree.
3. The size of the volume should be small enough that moving volumes between partitions is a viable approach to balancing server disk utilization and server load. Thus, volumes should be small relative to the partition size.
4. Finally, the size of a volume must not exceed the capability of the backup media.

A volume naming convention should also be used by those administrators who create volumes. Volume names are restricted to 32 characters and should be chosen so that given a volume name, a system administrator (who knows the naming conventions) can determine its correct location in the file system hierarchy. The convention used by the Coda project is to name volumes by their function and location. Thus, a replicated volume named "u.hbovik" is mounted in `/coda/usr/hbovik` and contains hboviks data. A project volume is prefixed by "p." and a system volume is prefixed by "s." Similarly, volumes containing machine specific object files are prefixed by the machine type. For instance, "p.c.alpha.pmax.bin" contains project coda binaries for our current alpha release and is mounted on `/coda/project/coda/alpha/pmax_mach/bin`.

Use the commands `createvol (8)` and **createvol_rep**(8) to create non-replicated and read-write replicated volumes respectively. (Read-only replication is discussed in Section 10.6 (XXX) below). These commands are actually scripts which ultimately invoke the `volutil (8)` command with the create option at the appropriate server. The volume will contain an access list initialized to **System:AnyUser rlidwka**. Creating the volume does not mount the volume within the file system hierarchy. Mounting the volume as well as changing the access list or the quota must be done using the `cfs (1)` command from a client. A new volume may not be visible at client workstations for some time (see Section 10.7 (XXX) below).

A few concrete examples should clarify the use of some of these commands. On the SCM, the command

```
* createvol u.hbovik mahler /vicepa
```

will create a non-replicated volume named "u.hbovik" on server "mahler"s /vicepa partition. Similarly, the command

```
* createvol_rep u.hbovik E0000107 /vicepa
```

will create a replicated volume named "user.hbovik" on each server in the Volume Server Group (VSG) identified by "E0000107". The file `/vice/db/VSGDB` contains the mapping between VSGs and their identifications. The names of the replicas will be "user.hbovik.n", where n is a number between 0 and |VSG| - 1.

10.3 Mounting a Volume

In order to use a volume which you have created and added to the appropriate databases, you must mount the volume. Although Unix file systems must be mounted upon reboot in Unix, Coda volumes are mounted only once. To mount a Coda volume, you must be using a Coda client and be authenticated as a user (use the **clog) command** who has write access to the directory in which the mount point will be created.

Mount the volumes using the command

```
* cfs mkmount <filename> volname
```

Note that cfs creates <filename> automatically. For example,

```
* cfs mkmount /coda/usr/hbovik u.hbovik
```

will create `/coda/usr/hbovik` and then mount the **u.hbovik** volume created in the example in Section 10.2 (XXX). The volume is now visible to all users of the Coda file system. When mounting a volume, avoid creating multiple mount points for it. Coda cannot check for this. More information about the **cfs** command can be found in Chapter 2 (XXX) as well as in Appendix E (XXX).

10.4 Deleting a Volume

When a volume is no longer needed, it may be purged by running the **purgevol** or **purgevol_rep** scripts on the SCM. Before removing a volume, you should probably create a backup for offline storage (see Section 10.6 (XXX) up to the restore step). The volume's mount point should be removed with the **cfs**(1) command (see the rmmount option) before purging the volume (if possible). Note that purging the volume will not purge related backup volumes. Backup and ReadOnly volumes should be purged with the **purgevol** script.

10.5 Restoring a Volume

For complete details on the backup/restore process, see Chapter 12 (XXX). In short, one first needs to get the correct dumpfile, possibly merging incremental dumps to an older full dump to get the desired state to be restored. Once this file is obtained, use the **volutil (8)** restore facility.

```
* volutil restore <filename> <partition> [<volname> [<volid>]]
```

The <partition> should be one of the `/vicep?` partitions on the server. You may optionally specify the volumeid and the volume name for the restored volume. This is useful when creating read-only replicated volumes. Note that currently dump files are not independent of byte ordering – so volumes cannot be dumped across architectures that differ in this respect.

10.6 Read-only Replication of a Volume

Read-only replication of a volume requires more effort on the part of the system administrator. However, it greatly increases the availability of volumes which cannot be read-write replicated. The most important example of such a volume is the root of the Coda file system. Conflicting updates cannot be allowed to occur at the root volume since this would make the entire Coda file system inaccessible. However, if the root volume is not replicated, the availability of the entire Coda file system depends upon the availability of the server acting as the custodian for this one volume. For these reasons, we highly recommend making the root volume of the Coda file system read-only replicated. We provide an extended example here to show you exactly how to go about the replication and distribution process. Note that the example shows how to make the root read-only replicated. Details pertaining to the coda root volume can be ignored when making other read-only replicated volumes (such as subtrees containing standard binaries).

We assume that you have a non-replicated root volume, called **coda.root**. If you are installing a new system, you can access this volume as /coda by using a Coda client. This volume should look exactly how you want the read-only replica to look. If it doesnt, make any changes now. Also note that any volumes that you want mounted within the read-only replicas should be mounted before continuing. Our root volume has three read-write replicated volumes mounted. The `usr` volume contains the home directories of our users, the `project` volume contains project directories and `tmp` contains temporary files. In addition, we have one subdirectory called `nonrep` which has non-replicated volumes (one per server) mounted within it. The purpose of these non-replicated volumes is to provide users with a location to perform repairs of conflicting objects. (Although the need for such a directory may not be clear at this point, we highly recommend providing such a directory.)

If you already have a read-only replicated root volume but want to update it, you should mount the read-write version of the root volume elsewhere and make your changes to this volume. Once you have made your changes, you will need to purge the old read-only replicas of your root volume using the **volutil(8)** command. Be sure that you purge the replica on each server. Then, you will need to edit the VolumeList file in `/vice/vol` and remove the entry for the read-only replicated root volume. (The name of the read-only replica will probably be coda.root.readonly.)

On the SCM, you need to clone the read-write copy of the root volume. You can use the command

```
* volutil clone <VolumeId>
```

This command will create a read-only volume with the name **coda.root.readonly** (Assuming that your root volume is called **coda.root**). Next, you will need to dump this cloned volume to a file with the command

```
* volutil dump <VolumeId> <filename>
```

Now, copy this file to each of the servers which will have read-only replicas of the root volume and execute the command

```
* volutil restore <filename> <partition> [<volid> [<volname>]]
```

Note that the root volume currently must reside on `/vicepa`. Read-only replicated volumes must share the same volid and name, so take care to specify these correctly when restoring to more than one server. The final step is to build the VLDB by running the command

```
* bldvldb.sh
```

on the SCM and to make sure that the file `/vice/ROOTVOLUME` contains the name or volume id of the root volume (coda.root.readonly). (Also, it may be necessary to restart the venus on the clients.)

10.7 Building the VLDB

The volume location data base, VLDB, is used to provide volume addressing information to workstations. Copies of the VLDB reside on the servers and are updated periodically. The VLDB lists the latest known location or locations of all on-line volumes in the system. A human readable version of the VRDB is maintained on the SCM in the file `/vice/vol/VRList`.

The VLDB is maintained on the SCM. When you wishe to update it, run the **/vice/bin/bldvldb.sh (8)** script on the SCM. The script gathers a copy of the `/vice/vol/VolumeList` file from all the servers, merges it into a single list, and builds a new VLDB. The UpdateMon program then propagates the new VLDB to all the servers. Note that the **createvol** and **purgevol** scripts automatically invoke **bldvldb.sh**.

10.8 Building the VRDB

The volume replication data base, the VRDB, is used to provide information about replicated volumes to client workstations. Copies of the VRDB reside on all servers and are updated periodically. The VRDB maps each logical volume to its corresponding set of physical volumes.

A human readable version of the VRDB is maintained on the SCM in the file `/vice/vol/VRList`. The **makevrdb** option to the **volutil(8)** command will create a new VRDB which will automatically be distributed to the other servers.

10.9 Building the VSGDB

The Volume Storage Group Data Base, VSGDB, is currently maintained by hand. Each valid volume storage group has an entry in this data base containing an identification number and the names of the servers in the group.

10.10 Ensuring Volume Consistency after Server Crashes

Coda servers ensure file system consistency after a crash by running **fsck (8)**, recovering RVM, and running the Coda salvager. The **fsck** used here a CMU has been modified so that it does not require every inode to be referenced by a [1] Coda accesses inode directly. After the server machine is booted, the **codasrv** process starts and RVM recovers the servers committed state. The Coda salvager then reconciles the results from **fsck** and the salvager.

10.11 Getting Volume Information

The **cfs** provides information on volumes. **cfs** can only be used on a machine which has a running *venus* (such as a client workstation). **cfs** is described in Chapter 2 (XXX) as well as in the manual page contained in Appendix E (XXX).

---

[1] **Warning**: the vanilla **fsck** must not be used on a Coda file system partition as the Coda files will be thrown away.

Chapter 11

System Administration: Users

11.1 Adding users

There are several steps in adding a new user to the Coda file system. First, create a Unix account for the new user on some or all of the Coda clients. Second, add the user to the Coda authentication database and the protection database (described below). Third, create a volume for the user and mount it at the appropriate place in the Coda name space (See Section 10.2 (XXX)).

11.1.1 Constructing new protection database files

Add a line for each new user to the `user.coda` file using the userId from the new users Unix account. @Foot(If you are at Carnegie Mellon and the user you intend to add does not have a CMU-CS account, STOP! The user **must** get a CMU-CS account before you may continue. To determine the UID, **grep** for the login name in `/etc/passwds`.) You may copy the user's entry in `/etc/passwds` to the user.coda file for this purpose if you wish. Note that only the login name (the first field) and the uid (the third field) are relevant and that you may delete the other fields as long as you leave all the "`:`" characters intact.

Now, add the users name to the appropriate group in the file `groups.coda`. Note that Coda groups are totally separate from Unix groups. Adding a user to group **System:Administrators** is about the equivalent of giving that user root privileges. Only System Administrators should be in this group. As an example, we add all Coda developers (hard and soft core) to the **system:coda** group. Add other Coda users to groups as appropriate.

Execute `% pwd2pdb -u /vice/db/user.coda -g /vice/db/group.coda > /vice/db/vice.pdb`

to generate a Coda protection database.

Finally, execute `pcfgen /vice/db/vice.pdb` to generate the file `vice.pcf`.

11.1.2 Installing the protection database files

Now that you have generated the new protection database files, you must install these files. Follow these steps:

1. Login into the SCM.
2. `cd /vice/db`
3. `/vice/bin/mvdb <srcdir> vice.pdb vice.pcf`) where <srcdir> is the name of the directory containing the new vice.pdb and vice.pcf.

11.1.3 Modifying the authentication database

User accounts are added to the authentication database with the **au** program, which can also delete and modify user accounts, change passwords, and get tokens. To add a user, run `au -h <SCM> nu` on a Coda client workstation, giving the System Control Machine's name in <SCM>.

In response to the prompts, provide your Coda userId and password and then the new users name and temporary password. (Give the new users full name to the "Other info" prompt.) You must be a member of the system:administrators group in order to successfully modify the authentication database. The following example illustrates adding the user "guest" to the authentication database. Note that the new password will echo to the screen.

```
au -h your-scm.host nu
Your Vice name: raiff
Your password:
RPC2_Bind () --> RPC2_SUCCESS
Vice user: guest
New password: guestpwd
New info: Guest User
```

Updates made via **au** will be automatically distributed to the other servers by the Update daemon. Now, the **auth** server knows about the new users. New users may change their temporary passwords using the **cpasswd** command.

11.2 Monitoring Auth Server Activity

The file `/vice/auth2/AuthLog` has a log of the auth server activity and can be used to monitor failed login attempts. You can also monitor password changes in the file `/vice/db/auth2.pw`.

Chapter 12

The Backup System

12.1 Introduction: Design of the Coda Backup Subsystem

As the use of the Coda file system increased, the need for a reliable backup storage system with a large capacity and a mininal loss of service became apparent. A one operation backup system was determined to be infeasible given the volume of data in Coda, the nature of a distributed filesystem, and the long downtime that would normaly be required to backup the system in one operation.

In order to meet the goals of high availability and reliability inherient in Coda design and to make efficient use of backup hardware and materials, the volume was chosen as the unit of data, and 24 hours was choosen as the time unit for system management and administration. The result of these design considerations is a volume by volume backup mechanism that occurs in three phases:

cloning

The cloning phase consists of freezing the (replicated) volume, creating a readonly clone of each of the replicas, and then unfreezing the volume. This allows mutating operations on the replicated volume to occur while maintaining a snapshot to backup. Once the cloning phase has been completed, normal read-write services can be resumed without fear of data corruption due to mutating operations on an active file system.

dumping to diskfiles on a backup spool machine

The dumping phase consists of converting the read-only volume clones to disk images stored as regular disk files, on a spool machine. A dump can either be full, in which all files are dumped; or incremental, in which only those files or directories which have changed since the last successful backup are included in the dump. This allows for a system in which only a subset of volumes need a full backup at anyone time (with incrementals done between full backups), thus reducing the amount of offline storage and network bandwidth needed at any one time. However, it allows the re-creation of data a granularity of 24 hours when combined with incremental dumps. Incremental dumps, however, are only supported for replicated volumes. Since there is little need for non-replicated volumes, only full dumps are supported for non-replicated volumes.

saving to media

The last phase consists of writing all the dump files from the backed up volumes dumped on local partitions to an archival media such as tape. Any standard backup system can be used for this phase. At CMU, we use the BSD dump and restore utilities to write and retrieve the disk images of Coda volumes to tape.

Practically, this system has been implemented as series of tasks The first two tasks are carried out by the backup program the latter by a Coda independent perl script (tape.pl).

- Saving:
  1. create a readonly clone
  2. dumping the readonly clone to a local disk
  3. backing up the dumped data to a suitable archive media
- Restoring:
  1. Retrieving appropriate full and incremental dumps from the archive media.
  2. Merging the full and incremental dumps to the time line of restoration.
  3. restoring the fully intregrated backup to the Coda file system.

Remeber, in practice, many restores are a result of a user accidently deleting or corrupting thier own files. In this case, users may use the the `cfs` mechanism to retrive files for the last 24 hour time line. For example:

```
cfs mkmount OldFiles u.hmpierce.0.backup
```

will mount the hmpierce's user backup volume from replica 0 to the OldFiles mount point. The file can then be copied out (backup volumes are read-only). Only if restoration needs are older than 24-hours or some chatastrophic event outside of the users control occurs do restores from tape normally need be done

Several tools have been developed to help in the creation, analysis, and restoration of data backups. Some of these tools have been developed by the Coda team (those tools concerning Coda FS to local disk conversion) such as `backup` and `tape.pl` (used to coordinate the efforts of `backup`, `dump`, etc), others employ off-the-shelf software such as the traditional UNIX **dump** or **tar** to tranfer the disk images created from the dump phase to the backup media. Coda, however, provides a perl script frontend to **dump**.

12.2 Installing a Coda Backup Coordinator Machine

The Coda Backup coordinator should be a trusted machine. It should be able get all the files that exist in the `/vice` subtree on the servers, although it is not necessary to run a fileserver on the backup coordinator (nor is it recommended that the backup machine be a fileserver).

Assuming the Backup Coodinator has been setup with an appropriate operating system, the steps are as follows:

NEEDS SECTION RECOMMENDING DATA/LOCAL DISK SPACE RATIO

On the Coda File Server designated as the SCM, create a file called `/vice/db/dumplist`. The dumplist contains three fields: volume id specified as its hex value, the full/incremental backup schedule, and a comment which is generally the human readable volume name. For example:

```
7f000001        IFIIIII        s:coda
7f000002        IIIIIFI        u:satya
```

The first column specifies the volume id to be backed up, the second column specifies the backup schedule by the day of the week begining Sunday, and the third column is a comment, ussually the volume name in human readable form. So, the volume id 7f000001 is scheduled for a Full backup every Monday, and incrementals Tuesday through Sunday and from the comment, we know this is a system volume called "coda". Likewise, the second volume 7f000002 is scheduled for a Full backup on Friday with incrementals being done Saturday through Thursday and is a user volume called "satya".

On the SCM, modify `/vice/db/vicetab` to indicate which host is acting as the backup coordinator and which partitions on the backup server are to be used by the backup coordinator to store the dump files. On a tripily replicated system, `vicetab` might look like this:

```
tye             /vicepa         ftree  width=8,depth=5
taverner        /vicepa         ftree  width=8,depth=5
tallis          /vicepa         ftree  width=8,depth=5
dvorak          /backup1        backup
dvorak          /backup2        backup
dvorak          /backup3        backup
```

`vicetab`, in addition to listing information on the servers providing replicated data, must also include information on the backup coordinator with backup coordinator's name in the first column, the backup partitions in the second column, and the designation "backup" in the third column. The 4th column is not used for the backup sub-system. Please see the *vicetab(5)* man page for additional information.

Note: that the number of partitions available for dumping may be controlled by the system administrator. Because the volume of data may be both large and variable, the **backup** program intelligently decides where to store individual dump files based on size accross the specified backup partitions. The directories in the sample **vicetab**, are assumed to be seperate local disk partitions. An organized central symbolic link tree is created by the backup.sh script in the directory `/backup` that points to the actual files scattered accross the `/backup1, /backup2, and /backup3` given in this example.

1. On the Backup Coordinator, the backup binaries and shell scripts need to be installed. The following directories should be have been created under `/vice` upon the installation of the Coda backup package:

```
/backup
/vice/backup
/vice/backuplogs
/vice/db
/vice/vol
/vice/lib
/vice/bin
/vice/spool
/vice/srv
```

In addition, the file `/vice/UpdateMonitor` should be created once the update monitor is run for the first time. The primary binaries that should be installed under /vice/bin to get started are:

```
backup
backup.sh
bldvldb.sh
```

```
merge
updateclnt
updatesrv
updfetch
tape.pl
```

Once it has been verified that the backup system is installed, the files

```
/vice/db/hosts
/vice/db/files
```

on the SCM should be manually copied to the same location on the Backup Coordinator. These are needed the first time by the `updateclt` daemon the when it runs. Also, Coda currently relies on the BSD dump and restore command to manipulate the tapes. A copy of dump package should be installed on the backup coordinator. BSD dump is available for all UNIX and UNIX like operating systems we have sucesfully run Coda on. Please check with your OS Vendor if you need help obtaining a copy.

============================== Upon completion **backup** will print which volumes were successfully backed up, the volumes on which backup failed, and those volumes which were not specified for backup.

The **merge** program allows a system administrator to update the state seen in a full dump by the partial state in an incremental dump. This is useful when a user wishes to restore to a state that was captured by a full and some number of incremental dumps (For instance, in the middle of the week) The **merge** program applies an incremental to a full dump, producing a new full dump file.

An incremental is a partial snapshot *with respect to* the previous dump. The Coda backup facility maintains an order on dumps for a volume. The merge program will only allow an incremental to be applied to its predecessor in the order. This predecessor may be a full dump or the output of the **merge** program.

Once the administrator has created or retrieved the full dump which contains the desired state of a volume, she can create a read-only copy of that state by using the **volutil restore** facility. This **volutil** command creates a new read-only volume on a server. The new volume can be mounted as any other Coda volume. Regular Unix file operations can then be used to extract the desired old data. The obvious exception is that mutating options will fail on files in a readonly volume.

12.3 Incremental Dumps

In every dump (full or incremental) produces a file containing the version vectors and *StoreIds* of every vnode in the volume. These files have names of the form `/vice/backup/<groupid>.<volid>.newlist` for replicated volumes and `/vice/backup/<volid>.newlist` for non-replicated volumes. When the backup coordinator is convinced that the backup of a volume has completed, the *.newlist file is renamed to be *.ancient via the **volutil ancient** call. These files are stored in a human-readable format for convenience.

When creating an incremental dump, the server looks for the .ancient file corresponding to the volume. If it doesnt exist, a full dump is created. If it does exist, it is used to determine which files have changed since the last successful backup. The server iterates through the vnode lists for the volume and the version vector lists from the **.ancient* file comparing the version vectors and storeIds. A discrepancy between the two implies that the file has changed and should be included in the incremental dump. Since version vectors are not maintained for non-replicated volumes, incremental dumps are not supported by the coda backup facility.

By comparing the sequence numbers in the vnodes lists, it can also be determined if a file or directory had been deleted (since the vnode is no longer in use). Vnodes that are freed and then reallocated between dumps look like vnodes which have been modified, and so are safely included in the incremental dump.

It is also important to maintain an ordering on the incremental dumps. To correctly restore to a particular day each incremental dump must be applied to the appropriate full dump. To ensure that this happens, each dump is labled with a uniquifier, and each incremental is labeled with the uniquifier of the dump with respect to which it is taken. During merge, the full dumps uniquifier is compared with the uniquifier of the dump used to create the incremental. If they do not match, the incremental should not be applied to the full dump.

12.4 Tape files

Once the dump files have been created, they must be written to tape. This is due to the fact that disk space is usually a limited commodity. The basic mechanism for the writing is the unix **tar (1)** facility.

Each tape contains a series of tar files, the first and last of which are labels. The start and end labels are indentical, and contain version information, the date the backup was taken, and an index which maps individual dump files into offsets into the tape. Thus the Coda backup tapes are self identifying for easy sanity checks. The label is a tar file which only contains a simple unix file called `TAPELABEL`.

The dump files are first sequenced by size. They are then broken down into groups, where the total size of the group must be larger than a certain size, currently .5 Megabytes. Each group is stored in a single tar file on the tape. These data tar files are the 2nd through n-1st records on the tape, the first and nth being a tar files containing just the tape label.

This structure was chosen for several reasons. The first is that it is easy to implement. Tar has been used for many years, and has been proven to be reliable. The second is easy access of information on the tape. Using a single monolithic tar file would often require hours of waiting to retrieve a single dump file. This way you can skip over most of the data using **mt (1)** and its fast-forward facility. Finally, it provides a simple and effective end-to-end check to validate that all the information has made it to tape.

At CMU, we have created a convention for capturing sufficient information for reliability, while trying to avoid excess use of tapes. Full backups are taken once a week. However, since our staging disks are not large enough to hold full dumps for all the replicas of all the volumes, we stagger the full backups across the week.

There are three kinds of requests for restorations: users who have mistakenly trashed a file, users who lost data but didnt know it, or bugs which require us to roll back to a substantially earlier state. The first class of restores can be typically handled by yesterdays state, which we keep on-line in the form of read-only backup clones. Thus almost all forms of requests never reach the system administrator at all. To give users easy access to the previous days backup, create a directory, OldFiles, in their coda directory, and mount each of the backups in the OldFiles directory.

If the user didnt catch the loss of data immediately, its reasonable to expect that they will catch it before a week has passed. We keep all incrementals and fulls to guarantee we can restore state from any day in the last week. This requires 14 tapes, or two weekly sets. One weeks worth is not sufficient, because state from later incrementals relies on earlier incrementals in order to be restored. Thus as soon as the first incremental tape is over-written (say Mondays), the state from the remainder of the last week is lost (last Tuesdays, Wednesdays, etc).

The third class of data loss is either due to infrequently used files or to catastrophy. (Weve actually been forced to rely on the backup system to restore *all Coda state due to major bugs in the servers*). Since its unreasonable to keep all the tapes around, we only save tapes containing full dumps. Weekly tapes are saved for a month, and monthly tapes are saved for eternity.

12.5 Restoring a backup clone

A basic assumption of performing backups is that eventually someone will need to restore old state of a volume. To do this they should contact the system administrator, specifying the volume (groupid and repid for replicated volumes or just the volid for non-replicated volumes) and the date of the state they wish to restore.

The system administrator must then determine which dump files contain the state. There could be more than one involved since the state may have been captured by a full and some incremental dumps. Once the administrator knows the dates of the backups involved, she must get the appropriate tapes and extract the dump files (via the **extract.sh script)**.

The administrator then creates the full state to be restored by iteratively applying the incrementals to the full state via the **merge** program. Once the state for the date in question has been restored, a read-only clone is created by choosing a server to hold the clone, and invoking the **volutil restore** operation, directing the call to the chosen server. Once the clone has been restored, the administrator should build a new VLDB, and mount the volume in the Coda name space so the user can access it. When the user has finished with it, she should notify the administrator in order for the clone to be purged.

12.6 Backup Scripts

Although the **backup** program handles all the tricky details involved in Coda backup, there still remains some issues to be handled, most notably the saving of the dump files to tape. This is done by a series of scripts, **backup.sh**, **writetotape.sh**, and **checktape.sh**. The job of extracting dump files from tape is handled by **extract.sh**.

backup.sh takes the name of the directory in which to run backups. It creates a subdirectory whos name indicates the date that backup was run. It then runs the **backup** program, using the dumplist file in the directory specified in the arguments, saving the output of **backup** in a logfile in the newly created subdirectory. It copies in the current Coda databases (so they will be saved to tape along with the dump files.) It then invokes **writetotape.sh** and **checktape.sh** to write and verify that the files have been safely recoreded.

writetotape.sh performs the work of saving the files on tape. It takes the directory in which the backup was taken (the subdirectory generated by **backup.sh)**, and the device name of the tape drive. It first checks to see that the tape to be used is either empty or has the correct label. For Coda at CMU this means checking that the tape was last used on the same day of the week. It then gathers the dump files and databases into groups and generates the tape label for this backup. Finally it writes the tape label and all the groups to the tape via the **tar (1)** facility, marking the end of the tape with another copy of the tape label.

checktape.sh verifies that **writetotape.sh** did its job correctly. Like **writetotape.sh**, it takes the backup directory and the name of the tape drive as input. It first reads off the tape label, comparing it with one stored in the backup directory. It then scans all the data tar files, comparing their actual contents with what it expected, and finally reads the tape label at the end, comparing it with the saved value.

extract.sh is used to extract a dump file from a Coda backup tape. It takes the name of the tape device, the date the backup was taken, and the identifier of the volume to be restored. The date should be specified in the form DDMMMYYYY, as in 10Feb1992. Volume identifiers are "groupid.repid" for replicated volumes and "volid" for non-replicated volumes. **extract.sh** will locate the correct group by reading the tape label, fast-forward the tape to the correct tape file, and extract the dump file for the volume.

Chapter 13

Reinitializing Coda Servers after a Disaster

As Coda is still an experimental system, the state of a Coda File System may sometimes get to the point where you will want to reinitialize all of the servers. *Note that this is a fairly drastic step and should be treated as a last resort.* It will typically take an entire day or more to perform and the system will be unavailable during that time. *You will* ***not*** *be able to reintigrate after a reinitialization.*

This chapter explains the steps necessary to reinitialize your system and restore your data on "clean" servers. The basic steps are as follows:

1. Obtain backup dump files for all of the volumes in the system.
2. Reinitialize RVM on the servers.
3. Recreate the volumes that existed before the reinitialization.
4. Restore and mount the backup clones created in step 1.
5. Mount the volumes created in step 3.
6. Copy the data from the restored volumes to the newly created volumes.
7. Unmount the new volumes and backup clones.
8. Flush client caches.
9. Test the system.

13.1 Obtaining Backup Dump Files

If the system is in a state that backups can be successfully run, we suggest running the **backup.sh** script to capture the most recent state. You may, however, use the tapes from your last successfule backup and skip this step. See Chapter 12 (XXX) and **backup (8)** for more details on creating backups. You will need to create new *dumplist* files for the backups. If you include the volumes that you do not normally back up in the newly created dumplists, you will be able to restore those volumes rather than re-create their contents. See **dumplist (5)** for details on the format of these files. To create the dumplists, use the following steps:

1. Use */vice/vol/VolumeList* and */vice/vol/VRList* to create a list of all of the volumes on the servers you want to reinitialize.
2. Sort the volumes by size. Use the **cfs lv** command to get an estimate on the size that each volume will take. Remember that if a volume is replicated, each replica will need space on the tape.
3. Divide the volumes into groups that will fit onto a single backup tape (at CMU this is about 1.5 GB)
4. Run **backup** on the dumplist files.

13.2 Reinitializing RVM on the Servers

Use **rvmutl** and **rdsinit** to initialize RVM. Unless you are reinitializing in order to change the size of the RVM heap, be sure to use the same sizes that you used previously. See section 7.4.1 (XXX) for the details of initializing RVM.

13.3 Recreate the Backed Up Volumes

Use **createvol (8)** and **createvol_rep (8)** to create all of the volumes that you backed up in section 13.1 (XXX).

13.4 Restore the Backup Clones

First, restore the read only coda of the *coda_root* volume. This should have been backup up in section 13.1 (XXX). In section 13.3 (XXX), you should have created your *coda_root.tmp* volume. This will automatically be mounted when *coda_root* is restored. Next create a directory */coda/tmp/restore* and use the **volutil restore** to restore each of the backup clones to */coda/tmp/restore*. See the **volutil (8)** man page for more details on restoring the volumes.

13.5 Mount the Recreated Volumes

Create a directory */coda/tmp/inited* and mount all of the volumes created in 13.3 (XXX) in this directory.

13.6 Populate the New Volumes

Run the script **restorevolumes (8)** to restore all of the volumes that were backed up in section 13.1 (XXX).

13.7 Unmount the Volumes

Use **cfs rmmount** to unmount all of the volumes in */coda/tmp/restored* and */coda/tmp/inited*. Then remove these directories. See the man page **cfs (8)** for more details on unmounting volumes.

13.8 Flush Client Caches

You need to flush the client caches as they will not have the correct volume information anymore. Use the **cfs flushcache** command to do this.

13.9 Test the system

After youve flushed the client caches the system is ready to run. Do some testing to make sure Coda is functioning correctly and allow the users back into the system.

Chapter 14

Coda Source Layout

This chapter is a guide to using and modifying Coda source code at CMU-SCS. It is mainly intended to help new members of the project come up to speed. But it may also help sites outside CMU that receive a Coda distribution to devise procedures appropriate to their environments.

14.1 The Big Picture

All files relating to Coda, with the sole exception of RCS files (see Section 14.8 (XXX)), are stored in `/coda/project/coda`. Let's call this `$CODA`. The directories under `$CODA` correspond to different *releases* of Coda, such as `alpha`, `beta`, and so on. A release is a complete set of *mutually consistent* sources, libraries and binaries. "Mutually consistent" means, for example, that the Venus and server from a release will work with each other. It also means that you can freely use the libraries and include files to compile new binaries in that release. @ux[There are no guarantees of any kind across releases.]

Before a new instance of a release is made, careful thought must be given as to how current users of that release will be affected. Will all such users have to be upgraded in one fell swoop? Or can they switch over at their own leisure? As the deployed Coda system grows, upgrading everyone in one fell swoop gets harder. After all, some of the users may be in hiding with their disconnected laptops!

By convention, the `omega` release corresponds to the version of Coda in production use. Most users Venii, as well as the servers on production machines, are from this release. Hence the value of `/usr/coda/SYMLINK` on those users workstations is `$CODA/omega`.

14.1.1 How versions evolve

The `alpha` release of software corresponds to a lightly tested version of the system. Individual developers have done a significant amount of testing of `alpha` software, but it hasnt yet been stressed heavily, nor has it been tested in real use. Hence `alpha` server code should not be run on servers that hold real user data. They should only be run on the `alpha` servers (also known as "test" servers). Similarly for other `alpha` software. When you are doing code development, you usually pick up include files and libraries from the `alpha` release. Thus, the private versions of software you build is, at best, of `alpha` quality.

The `beta` release corresponds to a version of the system that is expected to be released soon for production use. A small number of users, typically members of the Coda project, depend upon this release. Actual use by such users is part of the testing of the `beta` release. Eventually, when we have enough confidence in the stability and correctness of the `beta` release, we promote it `omega.`

`beta` is really a symlink to a volume mount point `beta-<unique>`, where `<unique>` is an identifier made up by the `makebeta` script described below. Upgrading the `beta` release consists of creating a new volume, mounting it at `beta-<unique>`, populating it using the steps described in Section 14.6.1 (XXX), and finally changing the `beta` symlink to point to `beta-<unique>`. The old `beta` volume is preserved for a while, until is clear that the new release hasnt triggered any serious problems. It can then be purged to reclaim space, or recycled for a future beta release.

`omega` is also a symlink to a volume mount point `beta-<unique>`. After a `beta` release has been running robustly for a while, it is upgraded to `omega` by merely changing the value of the `omega` symlink. No recompilation is involved.

When a `beta-<unique>` release is created, the `makebeta` script creates an RCS branch named `beta-<unique>` for every file in the release. Normally no development or changes are done along this branch. However, it provides a way to introduce an emergency fix in a `beta` or `omega` release if the need ever arises.

14.1.2 How code is developed

Lets illustrate this with a specific example. Suppose I want to upgrade the module `rpc2` and build a new Venus to use the upgrade. I begin by creating two private directories, say `/coda/usr/satya/src/rpc2` and `/coda/usr/satya/src/venus`, and populating them with source files from the `alpha` release of these two modules. Section 14.8 (XXX) tells you how this step actually gets done. Now I proceed to modify files in `rpc2` and to compile and test the module using standalone test programs. Then I modify files in `venus` to use the upgrade and then build a Venus. When building Venus, I must make sure that the version of `rpc2` used is the one I just built. Section 14.5 (XXX) tells you how to do this. Now I have a Venus that I can test. As bugs are found in the new Venus, I iterate the above procedure. When I am confident that my changes to `rpc2` and `venus` are right, I update the `alpha` release of these two modules.

The process of updating the `alpha` release is known as *installation.* Installation is the point at which work done by a Coda project member becomes visible to others in the project. Prior to this point, all work is done in that users own private directory. Installation always occurs at the granularity of entire *modules.* In other words, one never installs an individual library, include file, etc. Rather the entire source code for a module and all relevant files compiled from it are installed together. Modified files are automatically checked into RCS as part of the installation procedure. At some future time, these changes along with many others that were installed will make it into a `beta` release. Section 14.6 (XXX) shows you how this is done. After some use as `beta`, it will be promoted to `omega`.

14.2 Layout of Source Code

14.2.1 Layout of a Release

Each release has the structure shown in Figure 14.1 (XXX). Note that this layout is identical for all releases.

Underlined names are symlinks to machine-specific directories. On a 386 machine for example, `bin` is a

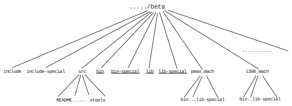

Figure 14.1: Structure of a Coda Release

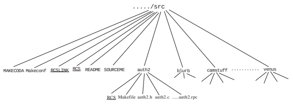

Figure 14.2: Structure of the source directory in a release

symlink to `i386_mach/bin`. Although only two machine-specific directories (`pmax_mach` and `i386_mach`) are shown, there can be many more.

The source tree for the release is in `src`. A copy of the header files from `src` is in `include`. Both `src` and `include` are machine-independent. For each supported machine type, there is a directory containing binaries (`bin`) and libraries (`lib`). The source tree in a release is fully self-contained. In other words, if you started out with empty `include`, `lib`, and `bin` directories, you could completely populate them by compiling the source code in `src`. The only exception to this are the files in `include-special`, `lib-special`, `src-special`, and `bin-special`. These directories contain a very small number of files that have to be copied in by hand. The sources for these files are not in `src`.

14.2.2 Layout of the source directory

The `src` directory in each release is organized as in Figure 14.2 (XXX). The `MAKECODA` script simplifies the compilation of the entire release. The `Makeconf` file defines in a single spot many key variables and paths used by the makefiles in individual modules. Those makefiles inherit these definitions automatically, when the CMU-SCS `make` is used. The `SOURCEME` file contains a minimal set of environment definitions. By sourcing this file before you compile anything, you can be sure that you arent obtaining binaries, libraries etc. from non-standard places. This is especially important if your `.login` or `.cshrc` files define elaborate `PATH`, `CPATH`, `LPATH` variables.

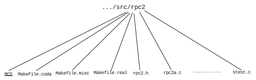

Figure 14.3: Typical module structure

Underlined names are symlinks to RCS directories. For example, `RCSLINK` is a symlink to `/afs/cs/project/coda/rcs`; `auth2/RCS` is a symlink to `../RCSLINK/auth2`; and so on. Indirecting via `RCSLINK` makes it simple to relocate the RCS directories without changing lots of individual symlinks. Such relocation might happen, for example, when Coda sources are used outside CMU.

14.2.3 Layout of a typical module

The structure of a typical module is shown in Figure 14.3 (XXX).

In this particular module, `rpc2`, the underlined name RCS indicates a symlink to `../RCSLINK/rpc2`. Indirection via `RCSLINK` makes relocating the RCS directories a simple matter.

Notice that only the source files are located in this directory; there are no object files. The CMU-SCS `make` facility puts all compilation targets elsewhere, thus allowing the source directory to be readonly. This has two advantages. First, the source directories are uncluttered. Second, it simplifies building Coda for multiple machine types, since the target directories can be different for each machine type.

Also notice the presence of multiple makefiles in this module. Although this is not characteristic of all Coda modules, it is typical of some. The true dependencies are capture in `Makefile.real`, and the others, like `Makefile.coda` and `Makefile.misc` invoke `Makefile.real` after defining environment variables appropriately. This simplifies the use of these modules outside Coda. If you are compiling by hand, you have to say `"make -f Makefile.coda <target>"`, rather than just `"make <target>"`.

14.3 Module Dependencies

Most Coda modules rely on files installed by other Coda modules. It is therefore important to install modules in the correct order. Otherwise you could get yourself into real trouble. If you are compiling from scratch, the out-of-order installations will simply fail. But if you are modifying an existing release, you could end up with mysterious bugs because obsolete versions of header files and libraries may be used.

You are @ux[strongly] urged to use the script `MAKECODA,` described in Section 14.4.3 (XXX). The correct precedences of modules are wired into the script, so you dont have to deal with them.

For the curious, the correct order of compilation of modules is given below. In principle, modules of the same precedence (i.e. in the same set) can be compiled in parallel. But I havent actually tried that yet.

1. `scripts` Miscellaneous scripts, including `alphaci` on which intalls of all other modules rely
2.
    - `mlwp` Lightweight process package
    - `dir` Directory package used by Venus and server
    - `sys` Miscellaneous routines
    - `sunrpc` Sun Microsystems public domain XDR code and interface for device driver/venus interaction
    - `igmp` Internet multicast support for old RFC (dummied out currently; someone should fix these to use the new RFC)
    -
3.
    - `util` Utility routines
    - `rpc2` RPC package
    -
4.
    - `camstuff` Header files that allow runtime choice of RVM or VM for persistence on servers
    - `blurb` Program to adjust copyrights
    - `pdbstuff` Protection database management
    - `rp2gen` Stub generator for RPC2
    - `comm` Communication layer above RPC2 for connection management (not yet in use)
    - `libal` Access list package.
5.
    - `vicedep` Header files and RPC2 interface definition files put here to break circular dependencies
    - `fail` Network failure and variable speed emulator
    - `auth2` Authentication server
6.
    - `login` Implements clog, cunlog, ctokens, etc.
    - `cfs` The VFS driver; most of this code is linked into the Mach kernel
    - `vv` Version vector routines
    - `mond` Coda usage data collector
7.
    - `resolve` Library used by repair (should get integrated into `repair`)
    - `vol` Volume package used by server
8.
    - `res` VM-based directory resolution algorithms
    - `repair` Repair tool
    - `venus` Cache manager
9.
    - `volutil` Volume utilities
    - `vtools` Miscellaneous Venus tools
    - `rvmres` RVM-based directory resolution algorithms
10.
    - `vice` Server code
11.
    - `update` The daemon which updates server databases
    - `norton` A Coda server, RVM debugger
    - `asr` Application Specific Resolver package
    - `egasr` ASR examples@end)annote

14.4 Building a Release

14.4.1 Prerequisites

To compile Coda "out of the box" you need the following compilation tools:

- Gnu g++ compiler version 2.5.8 or higher.
- CMU-SCS `make` The makefiles in the Coda sources exploit many of the enhancements made by CMU to Unix make. Conditional macro expansion, the ability to place targets in a different place from the sources, and the inheritance of variable definitions from Makeconf are the three major features we rely on.
- C compiler on IBM-RTs It is important that the compiler be set to use `long` to represent `enum`. The RT C compilers default behavior of using `char` to represent small-valued enums will cause insidious problems in RPC2 communication. On CMU-SCS machines this is done by defining the environment variable `PLAINC` to be `/usr/misc/.hc/bin/hc2`. The `MAKECODA` script does this automatically for you, but you must remember to do this manually if you arent using it. (Some `.login`s have the variable `ccC` defined to `hc`; you must delete this definition also).

It should be possible to modify the code to use other versions of C++ (such as `g++`), or to use standard Unix make. But we havent tried this, and dont plan to.

14.4.2 Special directories

Before you can compile Coda, you need to populate the `{include,lib,bin,src}-special` directories. These contain files that are (a) needed but arent in the Coda sources or (b) standard Mach header files, with slight modifications. The `MAKECODA` script contains an up-to-date list of what these files should be. Here is a list that was current at the time of writing this document:

RVM files

RVM is a lightweight transactional package that is used on servers and clients. It is a package that is independent of Coda. The current set of files from this package are:

```
include-special/{rvm.h,rvm_lwp.h,rvm_statistics.h,
        rvm_segment.h,rds.h}

lib-special/{librvm.a,librvmlwp.a,libseg.a,librds.a}

bin-special/{rdsinit,rvmutl}

src-special/{Makeconf,Makefile,READ_ME,plumber,rds,
        rvm,seg}
```

Tracing facility files

These files pertain to a file-tracing facility, `dfstrace`, used at CMU. To function as a trace-driven simulator, Venus requires the following files:

```
include-special/tracelib.h

lib-special/libtrace.a
```

If you don't intend to use Venus as a simulator, you could construct a dummy `libtrace.a` with empty routines to avoid unresolved references. You do need `tracelib.h` though.

Plumber

`malloc` The Coda makefiles allow you to build versions of `venus` and `codasrv` that use a special `malloc` to help detect memory leaks. The following files are needed for this:

```
include-special/newplumb.h

lib-special/{libplumber.a,libnewplumb.a}
```

If you don't plan to build the plumbing facilities, you can just create zero-length files with these names to keep the `MAKECODA` script happy.

Modified Mach headers

The changes in these files had to be made because of compilation errors from C++, or (as in the case of `assert.h`) to define different behavior for standard macros:

```
include-special/{assert.h,cthreads.h,setjmp.h,sys/inode.h,
        i386/fpreg.h,i386_mach/endian.h}
```

14.4.3 Using the MAKECODA script

Once you have taken care of the prerequisites and special files, you can compile a release. The simplest way to do this is to run the `MAKECODA` script in `src`. This script takes one required and two optional arguments. The required argument is `OBJECTDIR`, which is the pathname of the directory where the object files should be placed. It is sensible to specify a different directory for each machine type, and the `@@sys` facility of AFS and Coda lets you do this.

So, for example, to compile the `beta` release, I would do the following:

```
cd /coda/project/coda/beta/src
./MAKECODA OBJECTDIR=/coda/usr/satya/OBJS/@@sys
```

The `MAKECODA` script will first check to make sure that all necessary special files are present. If any are missing it will prompt you. It then goes through the Coda modules in the correct order and does a `make install` on each. The usual checkin to RCS that is done by `alphaci` as part of `make install` (see Section 14.5.1 (XXX)) is supressed. If all goes well, everything in Coda will be compiled, and the `bin`, `lib`, and `include` directores will be populated. You will have to repeat this once for each machine type.

Sometimes, you will run into a problem part-way through `MAKECODA`. After you have fixed the problem, youd probably like to continue where you left off rather than redoing everything from the beginning. Hence `MAKECODA` lets you specify the name of the module to start from. Here is an example:

```
./MAKECODA OBJECTDIR=/tmp/@@sys FIRSTMODULE=vol
```

Finally, the pathname of the root of the release you are compiling is specified by the variable `ROOT` in `MAKECODA`. You can change this by editing `MAKECODA`, and this is in fact what the `makebeta` script does for you when you create a new release. But you can also override it on the command line thus:

```
cd /tmp
MAKECODA OBJECTDIR=/tmp/@@sys  ROOT=/coda/project/coda/alpha
```

14.5 Making Incremental Changes

Once the `alpha` release has been built, you can start code development. We have already seen the general procedure in Section 14.1.2 (XXX). Lets look at an example in more detail.

Suppose I am working on the module `vtools` and need to change the files `cmon.c` and `codacon.c`. Heres what I would do:

```
# Create a scratch directory for my work
mkdir /coda/usr/satya/src
cd /coda/usr/satya/src

# Set up links to RCS directory, and root of release
ln -s /afs/cs/project/coda/rcs/coda-1.0 RCSLINK
ln -s /coda/project/coda/alpha/src SRCROOT
ln -s SRCROOT/Makeconf

# Create directory for this module
mkdir vtools
cd vtools
ln -s ../RCSLINK/vtools RCS

# Lock and checkout the file (s) to be modified
rcsco -l cmon.c codacon.c

# Now edit cmon.c and codacon.c

# Then compile this module
# Source the standard environment file to make sure your
# compilation environment (PATH, LPATH, CPATH, etc.) is set up right
source ../SRCROOT/SOURCEME
make OBJECTDIR=/coda/usr/satya/OBJS/@@sys cmon codacon

# Test cmon & codacon, then iterate on edit/debug cycle above

# Now you are ready to install your changes.
# You must first create a file called RCSMSG, and enter text
# in it that will become the RCS log message for the checkin.
# It will also be posted to the changelog bboard, so that
# others in the group will know of your installation

echo "Fixes to annoying bugs ... blah blah blah ..." > RCSMSG
make OBJECTDIR=/coda/usr/satya/OBJS/@@sys install
```

```
# You are now done!
# The install step automatically released the write locks on cmon.c
# and codacon.c

# Repeat the install step for each of the other supported platforms.
```

Notice that many other files may be need for compilation, but you dont have to check them out. This is because the CMU-SCS make knows to check out any needed files automatically into the compilation target area.

14.5.1 The alphaci script

The automatic checkin is done by a script called `alphaci` that is invoked as the last step of `make install`. `alphaci` will give you an error if the `RCSMSG` file is missing; it moves it to `RCSMSG.old` once it has checked in files. `alphaci` is smart enough to discover new files that arent mentioned in RCS, and prompts you to ask if you want them checked in too. Often you may not want this, because the files in question were just test files created for debugging. `alphaci` assumes that only writable files should be checked in; it prompts you about what to do with files that are not writable.

The RCSMSG file is also used by `alphaci` for posting on the changelog bboard (`cmu.cs.proj.coda.changelog` at CMU). You should follow the posts on this bboard closely, so that you are aware of changes to Coda modules by other project members. Here are some typical posts from the bboard:

```
03-Feb-93 18:56   M Satya   Installed scripts for IBMRT
Posted by alphaci from STRAUSS.CODA.CS.CMU.EDU
No files checked into RCS

03-Feb-93 18:50   M Satya   Installed scripts for I386
Posted by alphaci from WEBER.CODA.CS.CMU.EDU
No files checked into RCS

03-Feb-93 18:22   M Satya   Installed scripts for PMAX
Posted by alphaci from MOZART.CODA.CS.CMU.EDU
Files checked into RCS:      Makefile makebeta restartserver restore.sh
RCS message follows:
Created new script, makebeta, to make a clone of the entire alpha
source tree, and to create branches in RCS for all files.
```

Notice how the installations for machine types I386 and IBMRT caused no RCS files to be checked in. This is because the installation for the first machine type, PMAX, did the checkin.

14.6 Promoting Releases

As we have seen earlier, there are 3 important releases of software: `alpha`, `beta`, and `omega`. Servers are also classified by this scheme. At the time of writing this document, there were 3 `omega` servers (`rossini`, `puccini`, and `scarlatti`), 3 `beta` servers (`grieg`, `haydn`, and `wagner`), and 4 `alpha` servers (`schumann`, `gershwin`, `mahler`, and `vivaldi`).

The `omega` servers hold real data, so software let loose on them should have been tested very well. Having to reinitialize and to restore many gigabytes on the `omega` servers because of storage corruption is not an experience you are likely to forget! The `beta` servers hold some real data, but the number of users depending on that data is restricted to a few Coda project members. Further, that data is of a kind that can be easily reconstructed. The `alpha` servers are used by project members for testing. Obviously, only one person can test their software on a server at a time, so you should coordinate use of the servers with the other members.

14.6.1 Promoting alpha to beta

When the `alpha` release has diverged substantially from `beta`, and is relatively stable, we will decide to make a new `beta` release. One member of the Coda project will serve as *release coordinator* for the promotion.

The first step in this process is for the release coordinator to post on `cmu.cs.proj.coda.general`, asking project members to checkin changes and to drop all RCS locks on the main line of development by a certain deadline. As soon as possible you should release all mainline locks. It is ok to hold on to locks on private branches. Once the deadline expires, the release coordinator will feel free to break mainline locks.

In the second step, the release coordinator runs the `makebeta` script to create a clone of the current state of `alpha`. `makebeta` first synthesizes a unique identifier of the form *date_ xxxxx* where *xxxxx* is the number of seconds since midnight. The name of the new release is then `beta-<unique identifier>`; for example, `beta-3Feb1993_43696`. This name is used to tag a new RCS branch for every file in the release, so that emergency fixes are possible long after the `alpha` release has diverged from this `beta` release.

The `makebeta` script works as follows:

- First, it constructs a unique name for the release, as described above. It then pauses, waiting for the user to construct a directory with this name. Usually, the release coordinator makes this name the mount point of a new volume. `makebeta` checks to make sure the directory is empty before proceeding, to avoid accidental clobbers. For brevity, we refer to this directory as `$BETA`.
- Second, `makebeta` looks at `/coda/project/coda/alpha/src` (`$ALPHA/src`, for short) and determines the names of the modules in it.
- Third, it makes a copy of the `$ALPHA` modules in `$BETA`. In making this copy, it actually checks out the files (unlocked) from RCS, and compares them with `$ALPHA`. This is a sanity check, to ensure that RCS and `$ALPHA` are in sync. One side effect of this comparison is to detect RCS locks.
- If any locks were detected, or if any files in RCS differed from their copies in `$ALPHA`, `makebeta` pauses and asks the user to go take a look at whats wrong. It then asks the user whether to proceed.
- Next, `makebeta` creates a new RCS branch, named after this release, and re-checks out files from the branch so that `$Header$` in those files is correctly set.
- At this point, the files `MAKECODA` and `Makeconf` at the top level of `$BETA` are edited by `makebeta` so that all references to `$ALPHA` are changed to `$BETA`.
- Finally, `makebeta` copies over all the special directories (`{lib,bin,src,include}-special`).

The above procedure is quite slow, since many RCS interactions are involved. It usually takes a few hours. If all goes well, `makebeta` requires little babysitting. Note that the script does no locking. In other words, the `$ALPHA` tree better remain frozen for the entire duration of the promotion.

In the third step, the release coordinator `cds` into `$BETA/src`, and runs `MAKECODA` once for each machine type. Note that he doesnt have to specify anything other than `OBJECTDIR`, since `makebeta` has already set `ROOT` correctly in the `MAKECODA` script. This step guarantees that the binaries in `$BETA` were indeed compiled from the sources in `$BETA`.

In the final step, the release coordinator makes the symlink `beta` point to `$BETA`. This is the "commit" step that blesses the newly-cloned release as the `beta` release.

14.6.2 Promoting beta to omega

After the `beta` release has been stressed for some time, it will be promoted to `omega`. This is a much simpler procedure than the `alpha` to `beta` promotion. The release coordinator merely has to make the symlink `omega` point to the `beta-xxxx` where `xxxx` is the unique identity of the release being promoted. This "commit" step blesses what was hitherto `beta` as `omega`

14.7 Copyright Notices

Since Coda is distributed outside CMU, it is important that every source file contain a copyright notice. The `blurb` program simplifies adding and changing copyright notices. `blurb` expects to find the copyright notice is at the very beginning of a file, so make sure you dont move it when modifying a file.

The only modules without Coda copyright notices are `sunrpc,` which is public-domain code from Sun, and `cfs,` which is mostly Mach kernel code.

Some files in Coda are derived from the 1986 version of AFS-2.0. Since AFS-2.0 is owned by IBM, these files have an IBM copyright notice in addition to the Coda copyright notice.

When you modify a file or add a new file, you should pay attention to the copyright notices. Here are some simple rules to follow:

1. If you add new files, just include the Coda copyright notice. By definition, new code is not derived from AFS-2.0 and should not receive an IBM copyright notice.
2. If you modify an existing file, dont do anything with the copyright notice. All else being equal, add new code to new files, or to existing files without the IBM copyright notice. But use good judgement here – if it is clear that the correct home for a piece of code is a file with an IBM copyright notice, go ahead and put it there.
3. If you make a substantial copy of a file with an IBM copyright notice then the copy also acquires the IBM copyright notice. The interpretation of "substantial copy" is, of course, subjective. If you copy a variable declaration, it is not a "substantial copy". If you copy a whole bunch of procedures or macros, it is definitely a "substantial copy".

A simple way to think of this is that some files in Coda are "tainted" (i.e., they are derived from AFS-2.0). Tainted files can infect other files if enough of their innards are copied. Existing untainted files, and new files, stay untainted forever. In general, do your best to keep the number of tainted files to a minimum. Note that this discipline regarding copyrights is not intended to be onerous or constraining — just common sense and a little self-discipline.

14.8 Using RCS for Revision Control

1. Branch overview

One common problem in managing large software projects comes to play when there are several people working on the same sets of files. RCS helps by providing per-file locking to guarantee that no two users modify the same file at the same time. If several users are modifying the same line of development, however, this locking does nothing to guarantee that one users changes in one file wont interfere with another users changes in another file.

To combat this problem, RCS provides the notion of a "branch". A branch is a separate line of development carried out in parallel to the main line of development. Version found on branches have more than two digits in their version numbers. All digits make up the version number of an element on a branch. For example, version 1.2.1.1 is an element in the first branch off of version 1.2.

As branch numbers are hard to deal with, you are recommended to assign symbolic names to branches.

2. How to use branch

For example, a typical command to create and name a branch would be

```
rcsci -b -nKUDO_STATISTICS foo.c
        where KUDO_STATISTICS is a symbolic name of branch
* This command will create a branch named KUDO_STATISTICS
  off the last version on the mainline of foo.c
```

If you would like to specify the version number of which the first branch element is off, a command would be

```
rcsci -b -r1.2 -nKUDO_STATISTICS foo.c
        where 1.2 is the version number of which the first
        branch element is off
```

The name is assigned to the branch (not the branched element), so the subsequent commands such as

```
rcsco -l -rKUDO_STATISTICS foo.c
```

```
will refer to the latest element in that line of development.


3. What I checked in using branch

I checked in the following files using branch with the symbolic name
"KUDO_STATISTICS".

        vicedep/mond.rpc2
        venus/  fso.h
                fso0.c
                fso1.c
                hdb.c
                sighand.c
                venus.c
                venus.private.h
                venusresov.h
                venusutil.c
                venusvol.c
                venusvol.h
                venusvm.c
                venusvm.h
                vol_vsr.c
                vproc.c

The purpose of the modifications is to collect session statistics
and send them to the mond data collector. Currently, venii built
with these files are running on Brahms and faust, safely (I think so).
```

14.9 Coding Tips

<TO BE COMPLETED>

Appendix A

Quick Reference

To Disconnect If you need to set up yout hoard database, run:

```
% hoard -f <hoardfile>
```

for each project file.

If you already have your database set up, run: **hoard walk**.

cfs checkpointml	Checkpoint Coda modify log
cfs la	Display a Coda access control list
cfs sa	Set a Coda access control list
clog	Authenticate to Coda servers
cpasswd	Change your Coda password
ctokens	List Coda tokens
hoard	Set your hoard database
hoard list /dev/tty	List your hoard database
repair	Repair inconsistent files and directories
spy	List all Coda files that are referenced

Appendix B

System Files

File	Where	Created By	Edited By	Purpose
auth2.pw	/vice/db	initpw		Authentication server to generate secret tokens.
auth2.tk	/vice/db	hand		Used for secure communication among Coda servers.
dumplist	/vice/db	hand	createvol	Specifies which volumes to backup and when.
files	/vice/db	hand	hand	Update uses the files in files as a list of files to keep up to date.
groups_coda		hand	hand	By pwd2pdb to create vice.pdb file
hosts	/vice/db	hand	hand	Lists all of the servers in this Coda installation.
pro.db	/vice/db	adduser		Coda protection database
servers	/vice/db	hand	hand	Server number. Used as part of a volume number.
VLDB	/vice/db	buildvldb.sh		Volume Location Database. Used by clients to locate volumes.
vice.pcf	/vice/db	pcfgen	pcfgen	Auth2 binary password database
vice.pdb	/vice/db	pwd2pdb	pwd2pdb	Auth2 clear text password file
volutil.tk	/vice/db	hand		Used for secure communication among Coda servers.
VRDB	/vice/db	volutil makevrdb	volutil makevrdb	Map group volume names and numbers from a replicated volume to a VSG and the read-write volumes that make up the replicated volume.
VSGDB	/vice/db	hand	hand	By createvol_rep to map a VSG to a set of servers
AllVolumes	/vice/vol	buildvldb.sh		Human readable version of VLDB
maxgroupid	/vice/vol	hand	createvol_rep	By creatvol_rep when allocating group ids
partitions	/vice/vol	bldvldb		Lists vice partitions for all servers.
RWlist	/vice/vol	createvol		Lists all read write volumes.
user_coda		hand	hand	By pwd2pdb to create vice.pdb file
VolumeList	/vice/vol	createvol		List of volumes on server
VRlist	/vice/vol	volutil	volutil	Human readable version of VRDB.
SrvLog	/vice/srv	srv		Log of server activity.
SrvLog-xx	/vice/srv/	volutil		Old SrvLog.
UpdateLog	/vice/srv	update	update	Log of update activity.
AuthLog	/vice/auth2	auth2		Log of authenication server activity.
vstab	/usr/coda/etc	hand	hand	Venus configuration file Several other programs use vstab to get server names

Appendix C

Common Problems and Suggestions

`ls /coda` fails

If the return code is either "Invalid Argument" or "No Device", venus was probably restarted recently and hasnt yet finished initializing. Solution: Wait a few minutes a try again.

Newly created volume is

If a newly created volume, when mounted **dangling link**for the first time, shows up as a dangling symbolic link (it looks inconsistent), the problem is that new databases have not been distributed to the other replication sites. Solution: Unmount the volume, wait five minutes and try again. You'll have to reinitialize the venus which failed to find the volume, since vldb information cant be flushed.

Venus Fails on Startup with

There exists a process with an reference to a file in /coda. Its **"unable to unmount" message**either a running process **cd**-ed into `/coda` To solve the problem, either get the process out of /coda (in case its a shell) or kill it the process. If you cant find the right process rebooting the workstation should clear it up.

Venus error messages fail to print on console window of window manager

Use the *-console filename* option to **startcodav** on startup. This will force stderr messages to print to the specified file. Use **tail -f filename** in a window to observe any error messages.

Rvm gets Trashed

ABSOLUTE LAST RESORT If Rvm gets trashed – it asserts, wont recover, there are many failures on salvage, dumpvm fails – there are probably serious **venus** or **srv** bugs which must be fixed. Talk to Coda experts before continuing!!!

Performing the following actions will result in the loss of ALL Coda file system data.

Reinitializing the RVM Data segment (i.e. with rdsinit) Reinitializing the RVM Log (i.e. with rvmutl) Rewriting or trashing the file containing the Log or Data segment.

Appendix D

Example Configuration Files

D.1 /vice/db/user.coda

```
rootl:8eyUkE/DLFLOA:0:1:Local System Manager:/:/bin/csh
admin:A4SCSN80lqBao:2127:21:AccountsAdministrator:/usr0/admi
n:/etc/adm
jjk:trFEGNo9DEuN.:2954:32767:James Kistler:/usr0/jjk:/usr/cs
/bin/csh
satya:.Z.COf9BKQBPM:122:32767:M Satya:/cmu/itc/satya:/bin/cs
h
```

D.2 /vice/db/groups.coda

```
System:Administrators   -204    dcs jjk mre pkumar satya
System:coda             -221    dcs jjk mre pkumar satya
System:codastudents             -222    dcs jjk mre pkumar
```

D.3 /vice/db/servers

```
These server ids must not be reassigned.
Server id's must not be greater than 255.
Aliasing no longer allowed (i.e. do not map more than one name to one
number)

MAHLER.CODA.CS.CMU.EDU          201
VIVALDI.CODA.CS.CMU.EDU         202
RAVEL.CODA.CS.CMU.EDU           203
```

D.4 /.rhosts

```
bach
```

```
bach.coda.cs.cmu.edu
chopin
chopin.coda.cs.cmu.edu
debussy
debussy.coda.cs.cmu.edu
handel
handel.coda.cs.cmu.edu
mahler
mahler.coda.cs.cmu.edu
mozart
mozart.coda.cs.cmu.edu
pachelbel
pachelbel.coda.cs.cmu.edu
ravel
ravel.coda.cs.cmu.edu
vivaldi
vivaldi.coda.cs.cmu.edu
```

Appendix E

Unix Manual Pages

E.1 cfs(1)

NAME

cfs - Coda File System Interface Program

SYNOPSIS

cfs bandwidth <speed>

cfs checkpointml

cfs checkservers

cfs checkvolumes

cfs clearpriorities

cfs compress <file> [<file> <file> ...]

cfs [@cpu|@sys] cfs disconnect

cfs endrepair <file>

cfs examineclosure

cfs flushcache

cfs flushobject <obj> [<obj> <obj> ...]

cfs flushvolume <dir> [<dir> <dir> ...]

cfs getfid <path> [<path> <path> ...]

cfs getpath <fid> [<fid> <fid> ...]

cfs getmountpoint <volid>

cfs listacl <dir> [<dir> <dir> ...]

cfs listcache [<dir> <dir> ...]

cfs listvol <dir> [<dir> <dir> ...]

cfs lsmount <dir> [<dir> <dir> ...]

cfs mkmount <directory> <volume name> [-rw]

cfs purgeml

cfs reconnect

cfs replayclosure

cfs rmmount <dir> [<dir> <dir> ...]

cfs setacl [-clear] [-negative] <dir> <id> <rights> [<id> <rights> ....]

cfs slow <speed (bps)>

cfs truncatelog

cfs uncompress <file> [<file> <file> ...]

cfs waitforever [-on] [-off]

cfs whereis <dir> [<dir> <dir> ...]

cfs writedisconnect [-age <secs> -time <secs> <dir>]

cfs writereconnect [<dir> <dir> ...]

DESCRIPTION

cfs allows users to perform operations specific to the Coda File System. Most often, people use it to see how much of their allocated storage space they are currently using and to change the protection on their personal directories. **cfs** will change the protection or "rights" on directories but not on individual files. To change permissions on individual files, use *chmod (1)*.

bandwidth

Provide a hint about network speed. Use this command to tell venus your networks speed in bits per second.

Abbreviation: **bw**.

checkpointml

Checkpoint volume modify log. This command will create a checkpoint file `/usr/coda/spool/<uid>/<vol>@<mountpt>`. Where `<uid>` is your **cfs** user id, `<vol>` is the volume being checkpointed, and `<mountpt>` is the volume's mount point.

Abbreviation: **ck**.

checkservers

Check the status of the Coda file servers. Report on servers that are down.

checkvolumes

Check volume/name mappings.

clearpriorities

Clear short-term priorities used for cache management.

compress

compress given files in the cache. these files will be uncompressed automatically if they are touched or can be explicitely uncompressed with the *cfs uncompress* command.

@cpu or @sys

When the end of a pathname component is either @sys or @cpu, the local venus will replace these magic strings with a platform dependent string. These cfs commands can be used to check what the precise expansion values depending on the current OS/cpu.

disconnect

Partition your client from the Coda file servers.

endrepair

Tell venus to end a repair session on file. Useful if repair is exits without finishing the repair session.

Abbreviation: **er**.

examineclosure

Examine reintegration closure. Using **cfs examineclosure** will display the fixfile used to reintegrate changes that were made while operating in disconnected mode.

Abbreviation: **ec**

flushcache

Flush entire cache. Care must be taken when using the **cfs flushcache** command, as flushing the cache while operating in disconnected mode may result in loss of data.

flushobject

Flush objects from cache. **cfs flushobject** tells venus to remove the given objects from the cache.

Abbreviation: **fl**

flushvolume

Flush all data in specified volumes. Care must be taken when using the **cfs flushvolume** command, as flushing the cache while operating in disconnected mode may result in loss of data.

getfid

Map the given paths to Coda file ids.

Abbreviation: **gf**

getpath

Map fid to volume-relative path. **cfs getpath** will display the path for each of the given fids.

Abbreviation: **gp**

listacl

List access control list for each of the given directories.

Abbreviation: **la**

listcache

List the contents of the entire cache or the given volumes (directories).

Abbreviation: **lc**

listvol

Display the current status of the volume in which the directory is stored.

Abbreviation: **lv**

lsmount

List the countents of a mount point. This command can be used to tell what volume a mount point refers to.

mkmount

Create a mount point. Mount *volume name* at the point in the file system described by *filename*. If the *-rw* flag is set, never use the corresponding read-only volume for a volume. Otherwise, if a read-only volume exists (and the parent is also a read-only volume), use the read-only volume.

Abbreviation: **mkm**

purgeml

Purge volume modify log. Care must be taken when using the **cfs purgeml** command, as it may result in loss of data.

reconnect

Reconnect to the Coda file servers. **cfs reconnect** will undo the effects of a *cfs disconnect*.

replayclosure

Replay reintegration closure.

Abbreviation: **rc**

rmmount

Remove a mount point from the file system. The volume itself is not changed.

Abbreviation: **rmm**

setacl

Set access control list. Sets the access control list for each *id*. The *-clear* switch clears the access control list except for the entries given on the call to **cfs**. The *-negative* switch denies the given permissions, rather than granting them. *Rights* are a subset of *rwidlak* which are read, write, insert, delete, lookup, administer, and lock respectively. See the section on *File Protection* in the Coda manual for more detail.

Abbreviation: **sa**

slow

Set network speed to the given speed in bits per second.

truncatelog

Truncate the RVM log at this instant.

Abbreviation: **tl**

uncompress

Uncompress given cached files. *venus* will automatically uncompress any compressed files that it touches in the cache.

waitforever

Tells venus whether it should wait forever for dead file servers or not. By default, venus does not wait; it returns ETIMEDOUT. For certain batch jobs, waiting is better than not waiting.

Abbreviation: **wf**

whereis

List the servers that the given files reside on.

writedisconnect

Tell venus to write disconnect on the given volumes, or all volumes if none are provided. Write disconnected operation will fetch files from the server, but does not propagate changes back immediately.

An **-age** argument gives the age of the CML before it should be reintegrated. The **-time** arguments gives the number of seconds the sending of a reintegration fragment should take. Abbreviation: **wd**

writereconnect

Strongly connect to the servers.

Abbreviation: **wr**

SEE ALSO

chmod (1)

AUTHOR

1987, Adapted from AFS-2s fs

Maria R. Ebling, 1990, Created man page

M. Satyanarayanan, 1992, cfs rewritten from scratch

Joshua Raiff, 1993, Man page rewritten

Joshua Raiff, 1995, Updated

E.2 clog(1)

NAME

clog - Authenticate yourself to the Coda file system

SYNOPSIS

clog [[-x] user [password]]

DESCRIPTION

This command enables users to get tokens to use for authenticated communication with a Coda File System. The given name and password are passed to an *Authentication Server*. The *Authentication Server* returns a set of tokens encoding the chosen identity if the password is correct. **clog** passes these tokens to the *venus* process, which acts as the users agent for file manipulation. These tokens have a limited lifetime, becoming invalid after 25 hours. If an attempt is made to operate on a Coda File System object while the user is not authenticated, the user will assume the privileges of the *Anonymous* user.

clog accepts the following arguments:

user

Login name you wish to be identified as.

passwd

Password corresponding to the given user name. (If not provided on the command line, the user will be prompted for one.)

BUGS

The **-x** switch is not needed. Specifying a *username* and *passwd* is sufficient.

SEE ALSO

cunlog (1), venus (8)

AUTHOR

1987, Adapted from AFS-2

Maria R. Ebling, 1990, Created man page

E.3 cmon(1)

NAME

cmon - Coda server monitor

SYNOPSIS

cmon [**-a**] [**-t** probeinterval] server1:hz1 server2:hz2 ......

DESCRIPTION

cmon probes the specified list of servers once every *probeinterval* seconds and reports on their status. If a server is down or unreachable, **cmon** tries to reestablish contact with it once every *probeinterval* seconds. It uses the *curses (3)* package for screen management and can be used on dumb terminals. Run **cmon** in a terminal emulator (like *xterm*) if you are using a window manager (like *X*).

Hz1, hz2, etc. are the values of the kernel constant *HZ* on the respective servers. This constant is different for each machine type and denotes the number of jiffies per second. Common values are 64 (IBM-RT and DEC MIPS), 100 (Vax), and 50 (Sun3). This constant is needed to convert CPU usage (reported in jiffies by servers) to seconds.

Some **cmon** data can be displayed in relative or absolute modes. In relative mode, data is reported with reference to the interval between the last two probes. In absolute mode, the accumulated value since initialization is reported. **cmon** can be toggled between absolute and real modes of presentation from the keyboard. Typing `a` will cause data to be presented in absolute mode. Typing `r` will present it in relative mode. A mode change will only take place at the next probe.

The command-line options are:

-a

Report data in absolute mode.

-t probeinterval

Probe servers every *probeinterval* seconds. Default is 60.

The data reported by **cmon** is organized under four headings: *TIM, CPU, RPC,* and *DSK.*

The *TIM* data is as follows:

mon

time at which this **cmon** process was created.

prob

time at which the server was last probed.

succ

time at which the server last responded to a probe.

up

time at which the server process was started.

bind

number of times contact was reestablished after a failed probe. A probe may fail due to server or network failures.

The *CPU* data is as follows:

sys

relative or absolute number of seconds of system CPU time used on the server.

user

relative or absolute number of seconds of user CPU time (regular or niced) used on the server.

util

relative or absolute number of system and user seconds divided by corresponding time of accumulation.

The *RPC* data is as follows:

conn

number of RPC connections.

wkst

number of workstations connected to server. Note that each instance of **cmon** shows up as a workstation.

call

relative or absolute number of RPC calls received.

pki

relative or absolute number of RPC packets received. Includes duplicates and other bogus packets. Also includes bulk transfer packets.

pko

relative or absolute number of packets sent. Includes retransmissions. Also includes bulk transfer packets.

byi

bytes corresponding to *pki.*

byo

bytes corresponding to *pko.*

The *DSK* data is as follows:

max1

identity and percent usage of most full disk partition on server. The identity is the name of mount point. Names longer than 5 characters are truncated to the first 3 characters, a $ character, and the last character.

max2

identity and percent usage of second most full disk.

max3

identity and percent usage of third most full disk.

DIAGNOSTICS

If a server is down or unreachable, statisitics for it are reported as "???".

Relative data is undefined until two or more probes have been made. Such data is reported as "***" between the very first and second probes.

BUGS

Relative computations are just based on the difference between the two most recent probes. A smarter approach (especially for CPU utilization) would be to use some kind of weighted history.

The display is optimized for maximum packing of information into a small screen area. It may be cryptic for a novice.

No disk information is available about the root partition on a server.

Will core dump if run in a window with fewer than 25 lines.

No way to force a redisplay (eg ˆL).

SEE ALSO

vutil (8), curses (3), codacon (1)

AUTHOR

M. Satyanarayanan, 1990, Created

E.4 cpasswd(1)

NAME

cpasswd - change Coda login password

SYNOPSIS

cpasswd [-h host] [username]

DESCRIPTION

This command sets the password associated with the current user, or user *username* if explicitly named. Only the user or a System Administrator may change a password.

cpasswd prompts for the users old password (or the password of the administrator performing the change) and then for the new one. The new password must be typed twice, to forestall mistakes. Change requests can be aborted by typing a carriage return when prompted for the new password.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

cpasswd wont be able to change the users password if the password server is down. An appropriate error message will be printed, advising the caller to try again later. Once the password server acknowledges receipt the users new password, it takes about an hour for the information to propagate to the entire set of *AuthServers*.

cpasswd support the following option:

-h *host*

Tells **cpasswd** which authentication host to bind to. This should be the SCM.

FILES

/vice/db/user.coda

BUGS

The password file information should be kept in a different data structure allowing indexed access; *dbm (3X)* would probably be suitable.

SEE ALSO

clog (1), crypt (3)

AUTHOR

1987, Adapted from AFS-2

Maria R. Ebling, 1990, Created man page

E.5 ctokens(1)

NAME

ctokens - list all tokens

SYNOPSIS

ctokens

DESCRIPTION

The **ctokens** command enables users to query *Venus* for information about the set of authentication tokens it holds for the user. The **ctoken** command prints out the ID of the current user and whether or not the user is currently authenticated.

BUGS

If authenticated as another user, **ctokens** will say that you own the token.

SEE ALSO

clog (1), login (1), su (1), cunlog (1)

AUTHOR

1987, Adapted from AFS-2s unlog

David C. Steere, 1990, Created man page

E.6 cunlog(1)

NAME

cunlog - Tell the Coda Venus to release authentication tokens

SYSNOPSIS

cunlog

DESCRIPTION

This command enables a user to inform the Coda Cache Manager, *Venus*, to release the authentication tokens for that user. Without these tokens, the user will not be able to access or modify his or her protected files until he or she re-authenticates by running **clog**.

SEE ALSO

login (1), su (1), clog (1), ctokens (1)

AUTHOR

1987, Adapted from AFS-2s unlog

David Steere, 1990, Created man page

E.7 filcon(1)

NAME

filcon - the RPC2 filter control utility.

SYNOPSIS

filcon

filcon isolate -s srv1 [srv2 ...] **-c** clnt1 [clnt2 ....]

filcon join -s srv1 [srv2 ...] **-c** clnt1 [clnt2 ....]

filcon clear -s srv1 [srv2 ...] **-c** clnt1 [clnt2 ....]

filcon partition -s srv1 **-c clnt1**

filcon heal -h hostname1 port1 **-h** hostname2 port2

filcon oldpartition -h hostname1 port1 **-h** hostname2 port2

filcon slow -h hostname1 port1 **-h** hostname2 port2

filcon help

DESCRIPTION

filcon mainpulates the failure filters installed by low level RPC2 code. There are two basic modes of use, interactive and command line mode. The command line version are good for almost all common tasks but have less flexibility than the interactive version which gives full control over the filters.

The Coda failure emulation package consists of two pieces: a library of routines (*libfail*) that is linked in with each client and server, and a user interface program (*filcon*) that controls the behavior of the linked-in libraries. These components allow a wide range of failure scenarios to be emulated, including server failure, network failure, asymmetric and non-transitive communication, and lossy networks.

At each client and server, every incoming and outgoing packet is presented to the failure emulation package. Each packet is checked against an ordered list of filters. The first filter whose address matches the IP address on the packet (destination for *send* filters, or source for *receive* filters) is used to determine what action should be taken on the packet.

A filter consists of three parts: a *probability*, a *predicate*, and a *delay*. The probability specifies what fraction of packets that match the predicate are allowed to pass through unharmed. The *delay* may be used to delay packets in order to simulate a slow connection to those machines matching the filters address. The delay will apply regardless of the probability. Filters may either apply to *in*coming (receive) packets or *out*going (send) packets.

filcon enables a user to manipulate filters. There are two modes of operating with filcon: short summary **command line** commands, and **interactive use**. The interactive user interface presented by *filcon* is based on the *readline (3)* and Coda parsing libraries. A command with arguments can be issued in two ways. If only the command name is typed, *filcon* will prompt the user for the arguments. The command name may be uniquely abbreviated. Alternatively, all the arguments to a command may be given on the command line.

If any arguments are missing, the command will fail; there is no partial prompting. This feature is intended to allow the user to produce command scripts.

INTERACTIVE USAGE

Start this mode by typing **filcon**.

The following commands are available.

addclient ***host port***

Connects to a client at IP port number *port* on the host named *host*. (For Coda clients and servers, use 1363 and 1361 respectively.)

deleteclient ***clientnum***

Disconnects from; client number *clientnum*.

listclients

Lists the current set of connected clients.

insertfilter ***clientnum side id host color lenmin lenmax probability speed***

Inserts a filter into client number *clientnum*'s table on the specified *side* (one of *in out*). The filter will be inserted after the existing filter with identifier *id*. (Inserting after the filter with identifier *0* will insert the filter before all existing filters.) The characteristics of the new filter are *host, color, lenmin, lenmax, probability* and *speed. Host* is specified as a name or an IP address of the form "a.b.c.d." Any of the 4 IP components can be wildcarded by specifying a value of -1. Thus "-1.-1.-1.-1" matches any host, while "128.2.-1.-1" matches any host on subnet 2 at CMU. *Lenmin* and *lenmax* specify the minimum and maximum RPC packet sizes that match this filter. *Color* is a number in the range from 0 to 254. A value of -1 specifies that any color matches. RPC2 connections can be colored using the call RPC2_SetColor (). Packets sent out on a connection acquire the color of the connection. Note that reply packets acquire the color of the reply connection, which may be different. Colors are particularly valuable in stressing critical phases of higher-level protocols. By changing connection color between phases, one can easily investigate the effect of failures on specific phases. Probability is specified as an integer in the range from 0 to 10000, and correspond to floating point numbers in the range 0.00 to 1.00. The speed is the speed in bits per second that data should be transmitted between the machines. The value returned from this routine is the filters unique identifier. Speeds other than the default cannot be applied on *receive* filters.

removefilter ***clientnum side id***

Removes the filter with identifier *id* on the given side of client *clientnum.*

getfilters ***clientnum***

Lists the filters at client *clientnum.*

purgeFilters ***clientnum side***

Removes all filters from the set specified by *side*. *side* can be either *in*, *out*, or *both* (the default).

quit

Terminates the program. Note that this does not clear any filters.

COMMANDLINE INVOCATIONS

filcon isolate

isolate the servers and clients from all other Coda servers and clients. Servers will still listen to themselves allowing for system administration to proceed on the machine.

filcon join

This is normally issued after isolate and inserts a matrix of filters to allow communication between the clients and servers listed on the command line.

filcon clear

Removes all filter from the hosts given.

partition

blocks traffice between the hosts given.

heal

removes all of the filters between *hostname1* and *hostname2* inserted by partition, or slow, or filcon.

slow

inserts filters on *hostname1* and *hostname2* to control the network speed between the machines.

ARGUMENTS

port

Port to attatch to. This should be 1363 for a Coda client and 1361 for a server.

speed

Speed in bits per second

BUGS

You cannot currently designate a host by IP address of the form a.b.c.d as described above when binding to a process.

May confuse the user with the error message "RPC2_SUCCESS."

No authentication or security checks are done. Denial of service attacks are trivial, since any user can set up filters on any other client or server.

AUTHOR

Walter Smith, 1988, created. M. Satyanarayanan, 1990, updated. Puneet Kumar, 1990, Created Maria R. Ebling, 1991, updated. Lily Mummert, 1992, Added support for *speed*. David Steere, 1994, Added "purgeFilters", slight rewrite. Joshua Raiff, 1993, Created man page

Peter Braam, 1998, Complete rewrite for filcon.

E.8 hoard(1)

NAME

hoard - hoard database front-end

SYNOPSIS

hoard [-d] [-f source | cmds]

DESCRIPTION

Hoard is a front-end to the *hoard database* (HDB) managed by the Coda cache manager, *Venus*. The HDB provides a means for users to explicitly control Venus caching behavior. HDB entries specify the degree of a users interest in particular file system objects. Venus combines this information with implicit knowledge that it has about file access patterns to try to keep the "best" set of objects in its cache at all times. The HDB is maintained in non-volatile storage, so it survives Venus restarts and client reboots.

Users manipulate the HDB by issuing commands to the hoard program. The following hoard commands are currently recognized:

add filename <attributes>

clear <uid>

delete filename

list outfile <uid>

modify filename <attributes>

Hoard distinguishes between *children* of a directory, which are members of the directory, *descendants* which are either children or descendants of childres of the directory.

Commands may be abbreviated by their first letter. Parameters in angle brackets are optional, and have default values if unspecified. The *attributes* parameter is specified as a string of options separated by : characters. The currently recognized options are:

1-1000

assign this object the hoard *priority* indicated

c

current *children* of this directory will inherit its hoard status

c+

current and future *children* of this directory will inherit its hoard status

d

current *descendents* of this directory will inherit its hoard status

d+

current and future *descendents* of this directory will inherit its hoard status

If the *uid* in the clear and list commands is unspecified, the entries of all users are cleared or listed respectively. The default hoard *priority* is 10.

An example command file is:

clear

add /coda/project/coda/src 100:d+

add /coda/usr/jjk/.login 1000

Access to the hoard database is restricted in the following ways. All hoard commands fail unless the effective-uid is root–the hoard program should be made setuid to root to ensure this. Special permission is given to commands where the real-uid is root or that of the *primary* user, i.e., the user at the console. Root and primary users may add entries and access existing entries without restriction. Other users may not add hoard entries, and they may only clear, delete, list, or modify their own entries.

The command-line options are:

-d

Enables debugging output.

-f ***source***

Take commands from file *source*. -f must be the last argument if specified. An argument of - means use **stdin** as the source file. Source statements may be given directly on the command line (one per line) by enclosing them in single quotes.

DIAGNOSTICS

Hoard copies command lines that it cannot parse to **stderr**. If a syntactically correct command is rejected by Venus, the corresponding pioctl, its arguments, and the errno are copied to **stderr**.

BUGS

Negative priorities should be allowed.

A mount point of */coda* is assumed by the pioctl library.

SEE ALSO

venus (8)

AUTHOR

Jay Kistler, 1990, Created

E.9 mvdb(1)

NAME

mvdb - Atomically update files in a directory

SYNOPSIS

mvdb [-l lockfile] file1 file2 ...

or

mvdb [-l lockfile] newfile1=file1 newfile2=file2 .....

DESCRIPTION

mvdb is used to lock a directory exclusively and then atomically update a set of files in it. A file is moved in only if it does not already exist in the current directory, or if the version in this directory is older. You must be `cd`'d into the destination directory already. If you do not specify a *lockfile*, the current directory "." is locked. The Unix primitive *flock*() is used for locking. For each file moved in, you can choose to retain the original name or give it a new name by using the *newfile=file* construct. If a file *filename* already exists in the directory, it is first renamed to *filename.BAK*. The dates of the target are copied from the source files.

The options are:

-v

Verbose mode. Will print out information on the progress of the command.

-u

Force update. Moves in the files even if the source files are not newer.

EXAMPLE

```
cd /itcbin/vice/bin
mvdb /usr/andrew/bin venus2.sun=venus2 file salvage
```

BUGS

All the files have to come from the same source directory.

You have to be `cd`'d into the target directory. You need Lock rights in Vice directories for locking.

AUTHOR

Created: M.Satyanarayanan, 1985

E.10 repair(1)

NAME

repair - repair conflicts in the Coda file system

SYNOPSIS

repair [-d]

repair [-allowclear]

repair [<object_in_conflict> <fix_file> <repair_options>]

DESCRIPTION

The Coda repair tool allows you to manually resolve two kinds of conflict resulted from partitioned updates. Server-server conflicts are conflicting mutations performed on partitioned servers that can not be automatically resolved. Local-global conflicts are caused by mutations performed on a disconnected client that are in conflict with the global server state.

Repair uses the *ci* command interpreter to present its user interface or `repair` commands may be invoked from the command line. Several instructions may appear on a single line separated by semicolons. You can abbreviate command, variable, and help topic names. To send output into a file, end the command line with > filename.

To use the repair tool interactively, type **repair** at the command prompt.

Server-Server conflicts can be repaired from the command line without entering interactive mode. This is useful if you need to repair many conflicts within a volume at a time and wish to write a shell script. Please see the **EXAMPLES** section for examples on invoking complete repair sequences from the command line.

A description of the **repair** command follows:

beginrepair ***object***

verifies that *object* is indeed in conflict. It will print out messages to indicate whether the current repair session is for server-server conflict or local-global conflict.

For a server-server repair session, this command locks the corresponding volume and and mounts its individual replicas read-only. It will inform the users to only use the **comparedirs**, **dorepair** and **removeinc** commands to repair the conflict.

For a local-global repair session, both local and global replicas of *object* are visible at *object/local* (read-only) and *object/global* (mutable and serving as the workspace for storing the repair result for *object*). You need to iterate through the current sessions *local-mutations-list* containing all the local updates to *object* and its descendants. Each operation in *local-mutations-list* must be accounted for and Venus maintains the *current-mutation* being iterated. Use the **checklocal** command to find out the conflict between the *current-mutation* and the server state. Note that not all local mutations are necessarily in conflict, and you can use the **listlocal** command to see all the operations in *local-mutations-list*. You can advance the iteration to the next operation using either the **preservelocal** or the **discardlocal** command with the former replaying the *current-mutation* operation on the relevant global replicas. To speed up the iteration, the **preservealllocal** command repeats **preservelocal** until the *local-mutations-list* is exhausted or the first replay failure. Similarly, the **discardalllocal** command repeats **discardlocal** until exhausting the *local-mutations-list*. You can use external tools such as *emacs* to make direct updates on needed replicas under *object/global*. Use the **quit** command to either commit or abort the session.

quit

If the current session is repairing a server-server conflict, this command releases all the volume-level locks and causes the repair tool to return to the shell. If the current session is repairing a local-global conflict, this command asks you whether to commit or abort the repair session. If you answer yes, the mutations performed on the relevant global replicas will be committed to the servers. Otherwise, the repair result will be aborted as if this repair session has never happened.

help

prints out a help message.

Use the following commands to repair a server-server conflict:

comparedirs ***object fixfile [-acl <user> <rwlikda >] [-mode <newmode>] [-owner <username>]***

compares the mounted read-only replicas of a directory in conflict and prints the repair commands in fixfile to make the replicas identical. To print out the repair commands on your terminal give stdout as the pathname for fixfile. Compensating actions for Name/Name conflicts and Update/Update conflicts are not given. The user is only told about their existence and required to edit the fixfile manually. You should have already done a **beginrepair** on *object* and this command works only if *object* is a directory.

dorepair ***object fixfile***

does the actual repair of an object. If the repair succeeds, each accessible replica will be marked consistent. You will be prompted for the arguments if they are missing, and will be asked to confirm the repair. You should have already done a **beginrepair** on this object (or on on some other object in this volume.). If *object* is a file or symbolic link, *fixfile* must provide its new contents. If *object* is a directory, *fixfile* must provide a sequence of directory repair commands for each replica. The format of fixfile for directories is as follows:

```
replica <servername> <id of replica1>
    <repair commands for replica1>
replica <servername> <id of replica2>
     <repair commands for  replica2>
and so on
```

Repair commands are given one per line. Blank lines are ok. *id of replica1*, *id of replica2*, etc. are numbers that identify each replica. These are the same as the volume ids of read-write volumes corresponding to a replicated volume. The volume ids can be obtained by doing an ls on the inconsistent object, after the **beginrepair** command has succeeded. The directory repair commands are:

```
createf <filename>    <fid.0> <fid.1> <fid.2>
creates <symlinkname> <fid.0> <fid.1> <fid.2>
createl <linkname>    <fid.0> <fid.1> <fid.2>
created <dirname>     <fid.0> <fid.1> <fid.2>
removefsl  <filename or symlinkname or linkname>
removed  <dirname>
mv <srcname> <tgtname> <src <fid.0><fid.1><fid.2>>
                       <target <fid.1> <fid.2>>
setacl  <username> [rwlikda]
delacl  <username>
setmode <newmode>
setowner <new owner name>
setmtime <new modified time>
```

Note that for the **setacl** command, the short form access rights of **all** and **none** can also be used.

removeinc *object*

removes the inconsistent *object* if it is file or a symbolic link. If the object is a directory, all the descendants of the object will be removed in all the accessible replicas and the directory itself will be removed as long as its replicas are identical. If the owner or the ACL of the directory replicas are different, you have to repair the conflict first.

clearinc *object*

compares the mounted read only replicas of a directory in conflict and if the replicas are identical it clears the inconsistency flag of the replicas. Otherwise it will inform you about the inequality of the replicas. You should run the **comparedirs** command to find out the cause of conflict. This command should be used only for directories. Files and symbolic links are cleared of their inconsistency with the **dorepair** command.

The following commands are used only for repairing local-global conflicts:

checklocal

checks to see if the *current-mutation* being iterated by the current local-global repair session is in conflict with the server state. It displays the operator and operand (s) of the *current-mutation* operation and indicates whether it is in conflict with the relevant server replicas. If it is in conflict, a brief reason of the conflict is given. Note that this command does not advance the iteration of the *local-mutations-list*.

discardlocal

simply advances the iteration of the *local-mutations-list* of the current local-global repair session to the next operation. Use this command when the user does not want the *current-mutation* operation to have any effect on the repair result.

preservelocal

tries to replay the *current-mutation* of the current local-global repair session on the relevant global replicas. In other words, it will try to preserve the effect of the *current mutation* in the repair result. If the replay succeeds, the iteration of *local-mutations-list* will be advanced to the next operation. The effect of the replay is visible only on this client and not on the server until the repair result is successfully committed. If the replay fails, information about the reason of the failure will be displayed.

discardalllocal

repeatedly performs the **discardlocal** command until the *local-mutations-list* is exhausted. Its effect is to finish the iteration and discard the effect of all the remaining mutations on the repair result.

preservealllocal

repeatedly performs the **preservelocal** command until the first failure or the iteration of *local-mutations-list* is exhausted. This command is used if the user wants to preserve the effect of all the remaining mutation operations in the repair result.

listlocal

prints out all the mutation operations in the *local-mutations-list* of the current local-global repair session.

setglobalview

If the current local-global repair session is repairing a conflict on *object* given as the argument to the **beginrepair** command, the global replica of *object* is visible at *object/global*. This command allows the global replica of *object* to be visible at the original location (pathname) of *object*.

setlocalview

If the current local-global repair session is repairing a conflict on *object* given as the argument to the **beginrepair** command, the local replica of *object* is visible at *object/local*. This command allows the local replica of *object* to be visible at the original location (pathname) of *object*.

setmixedview

If the current local-global repair session is repairing a conflict on *object* given as the argument to the **beginrepair** command, this command restore the default form of local-global conflict representation in which the local replica is visible at *object/local* and the global replica is visible at *object/global*.

The following commands existed in old versions but are no longer supported:

showreplicas ***object***

shows the names of the individual replicas of *object*, and the pathnames by which these replicas may be examined read-only. A **beginrepair** must have been done earlier on this object (or on another object in the same volume).

unlockvolume ***pathname***

tells Venus to unlock the specified volume in repair. No check is done to see if you locked the volume during this repair session. The primary use of this command is to unlock volumes that were locked during a previous, aborted, invocation of the repair tool. The command will fail if Venus discovers that you do not hold the repair lock on the server. This could happen, for example, if your aborted repair occurred on another workstation, or if you were not repairing the volume in the first place.

EXAMPLES

This will cause repair to examine the object `"common"`, generate a fix file for it and in addition to the contents of the fix file, set the acl's for hmpierce on all of the replica.

```
repair common /tmp/fix -acl hmpierce all
```

The same repair would look like this in interactive mode:

```
repair> beginrepair common
repair> comparedirs common /tmp/fix -acl hmpierce all
repair> dorepair common /tmp/fix
repair> endrepair
repair> quit
```

SEE ALSO

ci (3)

AUTHOR

M. Satyanarayanan, 1989, Created

Puneet Kumar, 1991, Substantially revised

Joshua Raiff, 1993, Created man page

Qi Lu, 1995, Added local-global repair commands and revised man page

Henry M. Pierce, 1998, updated for command line options.

E.11 spy(1)

NAME

spy - Report all CFS files that are opened

SYNOPSIS

spy [-host <hostname>] [-uid <uid>]

DESCRIPTION

spy will connect to *venus* on the given host and report all Coda files that are opened. This is useful for generating *hoard* files as programs can have hidden dependencies. Using spy will ensure that any file that is opened gets included in the *hoard* database. Sending **spy** a *SIGTERM* will cause **spy** to flush all of its output buffers and terminate.

spy supports the following options:

-host *hostname*

Name of Coda coda client to bind to. If this option is omitted, then the current host will be used.

-uid *uid*

Only report on file activity from user *uid*. If the **-uid** switch is omitted, then all CFS file activity is reported.

To use **spy** to create a *hoard* file sort the output and remove any duplicates, By using **spy** you ensure that you did not forget to *hoard* a file with a hidden dependency. For example, when you *hoard* X, you will need the *uncompress (1)* program in order to use the fonts.

SEE ALSO

hoard (1)

AUTHOR

Joshua Raiff, 1993, Created man page

E.12 venussetup(1)

NAME

venus-setup setup **venus** on a client

SYNOPSIS

venus-setup [comma,separated,list,of,servers] [cachesize in kb]

DESCRIPTION

The **venus-setup** command takes a list of servers separated by commas, one of which must be the SCM and a cache size given in kilobytes. *NOTE*: at least one server must be specified. And if only one server is specified, it must be the SCM for the Coda Cell. **Venus-setup** then performs the following operations:

- Creates the */coda* mount point if not present.
- Creates */usr/coda* if not present.
- Creates necessary subdirectories under */usr/coda* if not present.
- Creates */usr/coda/etc/vstab* with parameters given on the command line.
- Creates */dev/cfs0* if not already present.
- Adds Coda filesystem port numbers to */etc/servers* if not present.

EXAMPLE

```
Venus-setup Mickey,Minnie,Goofy 10000
```

Sets up, */usr/coda/etc/vstab*, for **venus** to contact either Mickey, Minnie or Goofy and configures venus with a 10MB cache.

BUGS

The [cachesize in kb] option to venus-setup is not very smart. In fact, it is quite dumb. No abbreviations are allowed after the number and the number is taken literally to be kilobytes.

Venus-setup will happily overwrite an existing, */usr/coda/etc/vstab*, without warning.

FILES

/etc/services: the Internet network services list. */usr/coda/etc/vstab*: the VenuS TABle specification file.

SEE ALSO

venus (8), vstab (5), services (5), servers (5)

Coda Manual: “Installing A Coda Client”

AUTHOR

Henry M. Pierce, 1998, created

E.13 vicesetup(1)

NAME

Vice-setup setup a Coda server

SYNOPSIS

Vice-setup

DESCRIPTION

Vice-setup is the user front-end to a family of scripts designed to setup a Coda server based on question-answer responses. This avoids an otherwise tedious and error-prone-manual method. The most critical question asked for the vice-setup family of scripts to work properly is:

```
Is this the master server, aka the SCM machine? (y/n)
```

Answering "yes" to this question causes the following sequence of scripts to be called from within **vice-setup**:

- **vice-setup-scm**
- **vice-setup-user**
- **vice-setup-rvm**
- **vice-setup-srvdir**

where as answering "no" causes only the following scripts to run from **vice-setup**:

- **vice-setup-rvm**
- **vice-setup-srvdir**

OVERVIEW

Vice-setup

is designed to be called directly by the administrator to setup a server. It performs the following tasks common to both SCM and non-SCM servers:

- Creates the directories, */vice,/vice/*{*backup,db,srv,vol*} and, */vice/vol/remote* if they do not already exist.
- Creates and sets up the authentication files needed for interserver and interclient communication.
- Creates and sets up, */vice/db/files*, which are a set of common Coda files distributed from the SCM via, **updateclnt**, and **updatesrv**.
- Adds Coda port numbers to, */etc/services* if not already present.

- Sets up the Coda server to start at system startup if so indicated.
- Records the hostname in, */vice/hostname*.

Vice-setup-scm

This script is only run if “yes” is given as an answer to the SCM question.

- Prompts for a Coda server ID in the range of 0-255, and creates the file, */vice/db/servers*, with this information.
- Creates, */vice/db/scm*, with the hostname of the SCM machine being setup.
- Creates, */vice/db/VSGDB*, with the root volume number “E0000100”.
- Prompts for the name of the root volume and stores this information in, */vice/db/ROOTVOLUME*.

Vice-setup-user

This script is only run by, **vice-setup**, when the machine is designated as the SCM.

- This sets up the System:Administrator group and Coda user “admin” then initializes, */vice/db/vice.pdb*.
- Creates, */vice/db/passwd.coda*, needed to setup the initial password system.
- Creates, */vice/db/auth.pw*.

Vice-setup-rvm

- Prompts for the RVM_LOG device.
- Prompts for the size of the RVM_LOG device.
- Prompts for the RVM_DATA device.
- Prompts for the RVM_DATA size based on default values. These values must be typed exactly right to be accepted. For example, 22M must be typed exactly as, "22M".
- Initializes the RVM_LOG and RVM_DATA devices based on the values given for RVM initialization.
- Creates, /vice/srv.conf, with the values given for RVM.

Vice-setup-srvdir

This script sets up the data storage area for a Coda service.

- Prompts for the location a data storage area (default is, /vicepa).
- Creates the empty file, */vicepa/FTREEDB*, which must exist in order for, **makeftree**, to function.
- Creates, */vice/db/vicetab*.
- Initializes the data storage area with, **makeftree**.

For a detailed explanation of each question asked by the scripts, please see the chapter, “Installing a Coda Server” in the Coda Manual.

BUGS

Many of the highlights are:

- **Vice-setup-scm**: does not actually create the ROOTVOLUME it prompts for. It only sets up the accounting files that point the server(s) to the root volume. The actual volume must be created after the server is started for the first time with, **createvol_rep**.
- **Vice-setup-rvm**: will not warn you if run a second time after setting up an otherwise working server. This essentially causes all data stored on the Coda server in question to be wiped out.
- **Vice-setup-user**: there is no flexibility in setting up an administrative user called anything other than “admin”.
- **Vice-setup-user**: there is a security hole in, **initpw**, that could allow unauthorized Coda “admin” access and/or denial of service if an unauthorized user gains root access to the SCM.
- **Vice-setup-user**: a user “admin” with a uid of 500 must exist on each client that needs to have admin access to Coda. There is no practical way to test for this. The hardcoding of uid 500 may cause additional trouble at some sites.
- **Vice-setup-rvm**: will add entries to, */vice/srv.conf*, each time it is run without removing the previous contents. If more than one line of server information is present in, *srv.conf*, a “multiple instance error” is returned by, **srv**, because multiple lines in, *srv.conf*, are treated as a single set of command line arguments to **srv**. Essentially, the resulting error is made sufficiently out of context and is difficult to detect.
- **Vice-setup**: does not determine the host IP address.
- Error messages returned by the scripts or the programs called from within the scripts do not jump out and bark at you. Instead, errors are easily missed while running the scripts and these errors tend to come back and bite you later.
- If a non-SCM server is being setup, you still must respond with tokens to, **vice-setup**, when asked. However, the response must be identical to the SCM or the ticket files must be copied from the SCM manually for interserver communication to work correctly.
- **auth2**, **updateclnt** and **updatesrv** must be started manually on a non-SCM to suck down the real versions of these files from the actual SCM. This error in logic can be misleading.
- On non-SCM Servers, the following files on the SCM must be edited or copied from the SCM manually before running: **vice-setup**:*/vice/db/services*

  /vice/db/hosts

  to complete the setup of a non-SCM server.

FILES

/vice/db/vicetab: the Vice Table Configuration file for srv.

/vice/vol/VolumeList: volumeList of the server.

/vice/db/scm: the SCM hostname.

/vice/hostname: the host's hostname.

/vice/srv.conf: the **srv** configuration file.

/vice/db/services

/vice/db/ROOTVOLUME

/vice/db/VSGDB

/vice/db/vice.pfc

/vice/db/vice.pdb

And many more files are touched by these scripts than are listed here.

SEE ALSO

srv (8), rvmutl (8), rdsinit (8), auth2 (8), authmom (8), updatemon (8), updatesrv (8), updateclnt (8), startserver (8), srv.conf (8), createvol_rep (8), pwd2pdb (8), initpw (8), makeftree (8), vicetab (5)

Coda Manual: "Installing A Coda Server"

The RVM Manual

AUTHOR

Henry M. Pierce, 1998, created

E.14 volmunge(1)

NAME volmunge - manipulates objects within Coda.

SYNOPSIS volmunge <-adfmov> dir

-a

Prints out everything but does not cross volume boundaries.

-d

Prints out UNIX directories, but not volume mount points.

-f

Prints out all objects which are not volume mount points (eg UNIX files and UNIX directories); this preforms a stat() call on all non-volume objects which is ideal for forcing resolution on a volume.

-m

Prints out those objects which are volume mount points.

-o

Prints out those objects which are UNIX files and preforms an open() call on the file.

-v

Prints out the volume name of a volume's mount point.

DESCRIPTION

volmunge: is ideal for identifying Coda objects versus regular UNIX files (including UNIX directories) stored within the Coda filesystem. It will work recursively.

Because the, **-f** and **-o**, call the stat(), and open() functions respectively, resolution many be forced with volmunge if one is reconstituting a replicated Coda volume or group of volumes mounted on top of each other.

SEE ALSO

find (1), perl (1)

AUTHOR

Henry M. Pierce, 1998, created

E.15 histo(3)

NAME

histo - histogram package

SYNOPSIS

#include <histo.h>

InitHisto (hg, lolimit, hilimit, bucketcount, ht)

struct hgram *hg;

double lolimit;

double hilimit;

int bucketcount;

enum htype ht;

UpdateHisto (hg, newval)

register struct hgram *hg;

double newval;

PrintHisto (outfile, hg)

FILE *outfile;

register struct hgram *hg;

PlotHisto (outfile, hg, graphtitle, xtitle, ytitle, psfileprefix)

FILE *outfile;

struct hgram *hg;

char *graphtitle;

char *xtitle;

char *ytitle;

char *psfileprefix;

DESCRIPTION

This is a statistics package that can be used to produce the mean, standard deviation and histogram of a set of numbers. The histogram can be one of three types: linear, log2 or log10.

The package is used by calling the following routines:

InitHisto ()

initialises the histogram hg by setting the lolimit, hilimit, bucketcount, and histogram type as specified by the user. lolimit specifies the origin of the horizontal axis. hilimit specifies the maximum value on the horizontal axis. It is required that hilimit be greater than the largest value of the input data. The histogram type htype is specified as LINEAR = 1, LOG2 = 2, or LOG10 = 3.

UpdateHist ()

updates the histogram hg with newvalue to be entered.

PrintHisto ()

provides the user with statistics for the input data. Statistics include the mean, standard deviation, and the 90% confidence interval.

PlotHisto ()

the output of this file is used as input to the program csplot to produce a histogram of the input data.

DIAGNOSTICS

Negative return values indicate errors.

EXAMPLE

```
#include <libc.h>
#include "histo.h"
#define MAX 3


main ()
{


  int i, rc, bucketcount;
  struct hgram histogram1;
  double data[MAX], lolimit1,hilimit1;
  enum htype scale = LINEAR;
  FILE *fp1;

  data[0] = 5;
  data[1] = 15.4;
  data[2] = 10.6;

  bucketcount = 10;
  lolimit1 = 0;
  hilimit1 = 20;
  fp1 = fopen ("result1","w");
  if (fp1 == (FILE*) NULL) printf ("Cannot open\\n");

  rc = InitHisto (&histogram1, lolimit1, hilimit1, bucketcount, scale);
  if (rc < 0) printf ("Initialisation failed\\n");

  for (i = 0; i < MAX; i++)  UpdateHisto (&histogram1, data[i]);
```

```
    PrintHisto (fp1, &histogram1);

    fclose (fp1);


  }
```

AUTHOR

M.Satya, 1991

E.16 timing(3)

NAME

timing_paths - timing package

SYNOPSIS

#include <timing_paths.h>

#include <histo.h>

#include <dtcreg.h>

ti_init ()

ti_create (nEntries, thistie)

int nEntries;

struct tie *thistie;

ti_notetime (thistie, id)

struct tie *thistie;

long id;

ti_postprocess (thistie, twrt)

struct tie *thistie;

enum timewrt twrt;

ti_discoverpaths (thistie, pinfo)

struct tie *thistie;

struct pths_info *pinfo;

ti_stat (thistie, pinfo)

struct tie *thistie;

struct pths_info *pinfo;

ti_destroy (thistie)

struct tie *thistie;

ti_end ()

DESCRIPTION

This package may be used to time sections of a program. Time values are noted by placing probes in the code at points specified by the user. At each probe location a time value is obtained from a timer. These recorded time values are used to compute the times elapsed between consecutive probes. As an additional feature, for multiple runs of the same program, the package provides first and second order statistics for the elapsed time between consecutive probes. This package only works for timing experiments that take less than 35 minutes to run.

The package is used by calling the following routines:

ti_ init ()

initialises the timer.

ti_ create ()

allocates storage for n entries, where n corresponds to the total number of times ti_notetime () is called.

ti_ notetime ()

probes are placed in different sections of the code by calling ti_notetime (). Each probe has to be given a unique id number. In order for the program to function correctly it is necessary that a probe with id 0 is placed at the very beginning of the code being timed.

ti_ postprocess ()

the times obtained from all the ti_notetime () calls are processed by the ti_postprocess () routine. The ti_postprocess () routine can produce the time at each probe with respect to either the very first probe (twrt = BASELINE) or the previous probe (twrt = DELTA).

ti_ discoverpaths ()

for multiple iterations of the code being timed, different paths might be followed each time. A path corresponds to a set of ids beginning with 0, such as (0 1 2, 0 1 4). ti_discoverpaths identifies all the different paths traversed, the number of times each path was traversed and other relevant information.

ti_ stat ()

for each path traversed by the program, ti_stat provides the user with useful statistics (such as mean and standard deviation), for the elapsed time between probes.

ti_ destroy ()

frees the storage used by the package.

ti_ end ()

releases the timer.

DIAGNOSTICS

All successful calls return 0. Negative return values indicate errors. In particular -2 is returned by ti_postprocess if the experiment runs more than 35 minutes.

EXAMPLE

```
#include <timing_paths.h>
#include <histo.h>
#include <dtcreg.h>

main ()
{

  int i,j, foo, iterations = 10;
  int probes = 3;
  struct tie array_info;
  struct pths_info paths;


  ti_init ();
  ti_create (probes*iterations, &array_info);
  for (i=0; i<iterations; i++)
  {
    ti_notetime (&array_info, 0);
    ti_notetime (&array_info, 1);
    for (j=0; j<10; j++)
    {
      foo = foo +j;
    }

    ti_notetime (&array_info, 2);
  }
  ti_postprocess (&array_info, BASELINE);
  ti_discoverpaths (&array_info,&paths);
  ti_stat (&array_info,&paths);


  ti_destroy ();

  ti_end ();
```

AUTHOR

Gowthami Rajendran, 1991

E.17 backuplogs(5)

NAME

backuplogs.<DDMMMYY> - Format of a Coda backup log file

DESCRIPTION

A backuplog is created each day with day, month and year appended to the name "backuplogs". The file backuplogs contains the following information collected by the backup and BSD dump programs:

- the cloning process of each replica that makes up a Volume
- the dump of each replica to local tape
- Attempts to retry failed operations such as unable to contact a server
- Successfully volumes (incrementals are marked, otherwise full backups are assumed
- Lists any volumes that failed to be backed up
- Lists any partially backup volumes (e.g if a replica was missed)
- BSD Dump statistics
- the disk image file names dumped to tape

This file may be used to identify successful, partially successful, partially failed, or complete failures of a backup to occur. It is also used to identify the replica file names that make up a Volume to used by *merge*(8) to restore a replica. Also note that the BSD restore command is used to retrieve files from a tape listed in the backuplogs.<date> file. Restore should be done via restore(1) command provided by the BSD-DUMP facility. The the most recent incremental(s) are then merged with the most recent full backup of a volume.

EXAMPLE

```
date: Mon 01/05/98

00:05:26 Partition /backup1: 2561728 available size (1K blocks, minfree=5%), 256
1715 free blocks.
00:05:26 Partition /backup2: 2561728 available size (1K blocks, minfree=5%), 256
1715 free blocks.
00:05:26 Partition /backup3: 2559782 available size (1K blocks, minfree=5%), 255
9767 free blocks.
00:05:26 VLDBLookup: VLDB_size unset. Calling VCheckVLDB()
00:05:26 7f0003f4: cloning
00:05:29        e20000af->e20000b0
00:05:33        e10000bd->e10000be
00:05:36        e30000ae->e30000af
00:09:34 7f0003ef: cloning
```

```
00:09:57        e2000093->e20000ad
00:10:27        e10000a1->e10000bb
00:10:57        e3000092->e30000b4
00:29:02 Dumping 7f0003f4.e20000af to /backup1/05Jan1998/massenet.coda.cs.cmu.ed
u-7f0003f4.e20000af ...
00:29:02                Transferred 112131 bytes

00:29:02 Dumping 7f0003f4.e10000bd to /backup2/05Jan1998/poulenc.coda.cs.cmu.edu
-7f0003f4.e10000bd ...
00:29:03                Transferred 112131 bytes

00:29:03 Dumping 7f0003f4.e30000ae to /backup1/05Jan1998/rameau.coda.cs.cmu.edu-7f0003f4.e30000ae ...
00:29:14                Transferred 3768325 bytes
01:12:12 Dumping 7f0003ef.e2000093 to /backup1/05Jan1998/massenet.coda.cs.cmu.edu-7f0003ef.e2000093 ...
01:18:31                Transferred 119182513 bytes

01:18:31 Dumping 7f0003ef.e10000a1 to /backup2/05Jan1998/poulenc.coda.cs.cmu.edu-7f0003ef.e10000a1 ...
01:33:44                Transferred 119182513 bytes

01:33:44 Dumping 7f0003ef.e3000092 to /backup3/05Jan1998/rameau.coda.cs.cmu.edu-7f0003ef.e3000092 ...
01:42:03                Transferred 119172268 bytes

03:58:16                Transferred 309842 bytes

03:58:19
03:58:19 Attempting to retry any failed operations.
03:58:19
03:58:19 Successfully backed-up Volumes:
03:58:19 0x7f0003f4 (incremental)       f:u.satya2
03:58:19 0x7f0003ef                     f:u.mre
03:58:19 Only partially successfully backed-up Volumes:
03:58:19
03:58:19 Volumes that FAILED backup:
03:58:19
03:58:19 Volumes that were NOT backed-up:
03:58:19 0x7f000394                     t.test

[STATISTICAL SUMMERIES OF THE DUMP NORMALLY APPEAR HERE]

---------> Partition /backup:
---------> command: mt -f /dev/nst0 rewind
---------> command: restore -b 500 -s 1 -f /dev/nst0 -t /
Level 0 dump of /backup on dvorak.coda.cs.cmu.edu:/dev/sda1
Label: none
Dump   date: Mon Jan  5 04:05:14 1998
Dumped from: the epoch
         2      .
```

```
  11      ./lost+found
2041      ./05Jan1998
4081      ./05Jan1998/massenet.coda.cs.cmu.edu
4082      ./05Jan1998/massenet.coda.cs.cmu.edu/7f0003f4.e20000af
4087      ./05Jan1998/massenet.coda.cs.cmu.edu/7f0003ef.e2000093
6121      ./05Jan1998/poulenc.coda.cs.cmu.edu
6122      ./05Jan1998/poulenc.coda.cs.cmu.edu/7f0003f4.e10000bd
6127      ./05Jan1998/poulenc.coda.cs.cmu.edu/7f0003ef.e10000a1
8161      ./05Jan1998/rameau.coda.cs.cmu.edu
8162      ./05Jan1998/rameau.coda.cs.cmu.edu/7f0003f4.e30000ae
8167      ./05Jan1998/rameau.coda.cs.cmu.edu/7f0003ef.e3000092
```

SEE ALSO

backup (8), dump (1), restore (1), dumplist (5), vicetab (5), merge (8)

AUTHOR

Joshua Raiff, 1993, Taken from the System Administrators Guide.

Henry M. Pierce, 1998, updated.

E.18 dumpfile(5)

NAME

dump file - Format of a dump file

DESCRIPTION

The dump files are formatted as tagged byte streams. Each piece of the dump is marked by a tag byte, a number between 1 and 20. Each piece may also be broken into subpieces, each marked by a letter of the alphabet. Each subpiece is uniquely marked, but tags can be reused in different parts of the dump.

The first 60 bytes of the dump contain header information. This header info identifies which backup volume was used to create the dump, and when and from which volume the backup clone was created. It also indicates whether the dump was incremental, and provides information to be used later in placing this dump in the proper sequence for merging. Following the dump header is the volume information structure used by the Coda internals. Following this are two sequences of vnodes, one for directories and one for files. Each sequence consists of a header and a stream of vnodes.

The vnode sequence header contains the number of vnodes and the size of the vnode list. These numbers are not necessarily the same since not every list needs to have a vnode on it and some lists may have more than one. This is an artifact of the way vnodes are stored in the Coda servers.

After these two fields comes a list of vnodes. Each vnode consists of two parts, the first is the meta information associated with the file or directory and the second is the data for that vnode. This data could either be directory pages or file data. The first word after the tag for the file or directory data is a count of bytes or directory pages. For directory vnodes, the access list for that directory is included as part of the meta information.

SEE ALSO

backup (8), dumplist (5), volutil (8)

AUTHOR

Joshua Raiff, 1993, Taken from system adminstrators guide.

E.19 dumplist(5)

NAME

dumplist - List of volumes to be archived by the Coda backup mechanism

DESCRIPTION

The dumplist file contains information for the **backup**(8) program. It consists of a list of volumes to be archived.

Each entry in the list contains three tab separated fields: the volumeId (groupId), the schedule for full and incremental dumps, and a comment used to facilitate reading the output of the **backup** program. The schedule contains a number of "F" and "I" characters. Each character represents what form of dump to take for that day in the schedule. The length of the period is the number of characters in the schedule. For instance, "FIIIIII" means take one full dump followed by 6 incrementals, whereas "F" means take a full dump every day. For convenience, the first character in a 7 day schedule refers to Sunday.

EXAMPLE

The following does a full backup of *coda_root*, *coda_root.i386*, and *coda_root.tmp* on Monday while it does a full backup of *coda_root.project* and *coda_root.usr* on Wednesday. It uses */vicepa/backup* and */usr/codabackup/backup* to store the dump files.

```
7F000000        IFIIIII        coda_root
7F000001        IFIIIII        coda_root.i386
7F000002        IIIFIII        coda_root.project
7F000003        IFIIIII        coda_root.tmp
7F000004        IIIFIII        coda_root.usr
7F000005        IFIIIII        coda_root.misc
```

NOTES

In prior release of coda, the key words "Partition" and "Volumes" were used to distinguish between seperate sections of the dump file needed by `backup`. `dumplist` should now only include the Volume backup information and does not need the key word Volume. Nor is the keyword Partitions needed any more as `backup` reads vicetab for backup partition information.

SEE ALSO

backup (8), creatvol (8), createvol_rep (8), vicetab (5)

AUTHOR

David C. Steere, 1991, Created updated 1998, hmp

E.20 groups.coda(5)

E.21 maxgrpid(5)

NAME

maxgroupid - replicated volume number allocation mechanism

DESCRIPTION

The system control machine (SCM) has a file, */vice/vol/maxgroupid*, which contains the maximum replicated (aka group) volume number that has ever been allocated in the system. This is used as a simple way to guarantee that group volume numbers are unique.

Replicated (or group) volume ids are allocated out of the same namespace as replica ids and non-replicated volumes ids. The latter two types of ids have a 1 byte (8 bit) prefix to identify the server on which they are stored. We suggest using prefixes in the range 00-7F for replicated volume ids, and prefixes in the range 80-FF for other volumes. When initializing a system, put the number you wish to use for the first replicated volume in */vice/vol/maxgroupid*. For example, if you wish to use 7f000000 as the first replicated volume, create */vice/vol/maxgroupid* with the number 2130706432.

FILES

/vice/vol/maxgroupid on the SCM

BUGS

This file should be salvaged, but it is not.

Removing this file (without reinitializing the whole system) is a recipe for disaster. Grave confusion will result if group (or any other) volumes are created with the same number.

SEE ALSO

servers (5), createvol_rep (8)

AUTHOR

Jay Kistler, 1990, Created

E.22 mcstagen(5)

NAME

multicastagents - multicastagents file specification

DESCRIPTION

This text file lists all the multicast agents known to the system. It is read by each magent daemon to learn the IP addresses of its colleagues and the range of transient IP multicast addresses that it is allowed to allocate. Each line describes one multicast agent, according to the following format:

```
<agent address> <min allocation address> <max allocation address> <comments>
```

All addresses are in internet dot format (a.b.c.d). Allocation addresses must be valid class D addresses, i.e., in the range 224.0.0.0 to 239.255.255.255.

FILES

/etc/multicastagents

BUGS

This file is applicable only for multicast support as specified in RFC-988. A new multicast support specification, RFC-1054, has been released which supercedes RFC-988.

SEE ALSO

magent (8), multicastgroups (5), RFC-988

AUTHOR

Jay Kistler, 1990, Created

E.23 mcstgrps(5)

NAME

multicastgroups - multicastgroups file specification

DESCRIPTION

This text file lists all the permanent IP multicast addresses and their corresponding membership keys. It is read by each magent daemon. Each line describes one multicast group, according to the following format:

```
      <multicast address> <key-high> <key-low> <group name>
```

All addresses are in internet dot format (a.b.c.d), and must be valid class D addresses, i.e., in the range 224.0.0.0 to 239.255.255.255. Membership keys are represented as a pair of 32-bit hex numbers. Unrestricted groups should have null keys (i.e., 00000000).

FILES

/etc/multicastgroups

BUGS

This file is applicable only for multicast support as specified in RFC-988. A new multicast support specification, RFC-1054, has been released which supercedes RFC-988.

Only unrestricted groups should be used. The protection offered by restricted groups is bogus, and was never tested.

SEE ALSO

magent (8), multicastgroups (5), RFC-988

AUTHOR

Jay Kistler, 1990, Created

E.24 servers(5)

NAME

servers - map server names to numbers

DESCRIPTION

Each file server has a file, */vice/db/servers*, which maps server names to numbers. The server numbers are 8 bit numbers so a maximum of 256 servers can be supported. Once a server number has been assigned, it may **not** be reused. You should not reassign a server a new id without initializing the server.

The server numbers are used in too many places, probably the most important is in the creation of volumes. The server creating a volume uses its server number as the first byte of the four-byte volume identification number in order to ensure uniqueness.

NOTE: Replicated volume numbers must also be unique, so we reserve part of the range, 0-127 (0-7F hex), for this purpose. Real server numbers must **not** be allocated from the other subrange, and vice-versa.

The format of the file is the full server name (e.g. mahler.coda.cs.cmu.edu) followed by a tab and then the servers number. Any line beginning with a "\#" is considered a comment and is ignored. The following is an example servers file:

```
MAHLER.CODA.CS.CMU.EDU          201
GRIEG.CODA.CS.CMU.EDU           204
HAYDN.CODA.CS.CMU.EDU           205
WAGNER.CODA.CS.CMU.EDU          206
DEBUSSY.CODA.CS.CMU.EDU         212
SCARLATTI.CODA.CS.CMU.EDU       214
ROSSINI.CODA.CS.CMU.EDU         215
PUCCINI.CODA.CS.CMU.EDU         216
GERSHWIN.CODA.CS.CMU.EDU        218
SCHUMANN.CODA.CS.CMU.EDU        219
```

FILES

/vice/db/servers

BUGS

The server numbers and replicated volume prefixes are pulled from the same number space. Separation is enforced only by convention.

This file and /vice/db/hosts should be combined.

SEE ALSO

vrdb (5), vsgdb (5), maxgroupid (5), hosts (5)

AUTHOR

Maria R. Ebling, 1990, Created

E.25 user.coda(5)

E.26 passwd.coda(5)

NAME

passwd.coda - Coda user identification file

DESCRIPTION

This file specifies initial cleartext passwords for Coda users. The format of the file is:

```
uid<TAB>acutal cleartext password<TAB>any desired info
```

Where:

ViceId

is a Coda user id.

Cleartext Password

is the password to use for this *ViceId*.

Other user info

is other information about the user you want to include, e.g. the user name

WARNING this file must be owned by root and not be readable by any other user.

This file is translated into a Coda encrypted password file by initpw(8).

SEE ALSO

initpw(8), vice.pcf(5), pwd2pdb(8)

AUTHOR

Lesa Gresh, 1998, created

E.27 vicetab(5)

NAME

vicetab - Vice Filesystem Table

DESCRIPTION

vicetab is a table of Coda data partitions found on individual servers comprising a Coda netowrk server. This includes partition(s) used by the backup coordinator to store dump files to placed on a suitable backup media. This file must be shared among all machines comprising a Coda hub so edits should be done only on the designated SCM. For example:

```
tye /vicepa ftree depth=5,width=8 tye /vicepb ftree width=8,depth=5 taverner /vicepa
ftree width=8,depth=5 taverner /usr/vicepb ftree depth=4,width=8 tallis /vicepa ftree
width=8,depth=5 tallis /vicepb ftree width=8,depth=5 dvorak /backup1 backup dvorak
/backup2 backup dvorak /backup3 backup
```

where column 1 specifies the server as returned by the system call gethostbyname().

Column 2 specifies the directory of the Coda data tree which must be a local file system partition for optimal performance. **NOTE:***if a server serves than more than one Coda data parition, each data partition must have a seperate entry in vicetab.*

Column 3 specifies the Coda partition type.

- *ftree* an "inode number" directory and file naming system for storing the Coda filesystem into a local file system tree structure: e.g. inode 4711 in base 10 with width 1 and depth 5 would become 0/4/7/1/1. So, using the above example, a width of 4 and a depth of 2 would make the file name 4/711. If this storage type is specified, the width and depth must also be given in the 4th column seperated by a comma.

Column 4 specifies the width and depth of the ftree.

- *width* the maximum "width" of an ftree data partition.
- *depth* the maximum "depth" of an ftree data partition.

In the case of the backup coordinator, only the first three columns are used, with a partition type of `backup` specified in the third column.

This file should only be edited on the designated SCM machine and then allowed to propagate.

FILES

/vice/bin/makeftree

BUGS

None we currently are aware of.

SEE ALSO

makeftree (8)

AUTHOR

Henry M. Pierce, 1997, created

E.28 volulist(5)

NAME

volumelist - volumelist file specification

DESCRIPTION

Every server keeps a list of volumes for which it is the custodian. This list is in */vice/vol/VolumeList*. Every time a volume is created, an entry corresponding to that volume is made in this list. *bldvldb.sh*(8) uses this list to generate the Volume Location Data Base (VLDB).

The first few lines of this file contain information pertaining to the disk partitions. These lines have the following format:

P<partition-name> H<hostname> T<total usable space on this partition> F<free space on this partition>

There is one line entry for each volume on the server. Each line begins with W, R or B depending on whether the volume is read-Write, Readonly or Backup. The format is as follows:

R|W|B<volume-name> I<volume-id> H<server id> P<partition name> m<min quota> M<max quota> U<disk space used> W<parent id> C<creation date> D<copy date> B<backup date> A<volume usage>

FILES

/vice/vol/VolumeList

SEE ALSO

createvol (8), createvol_rep (8), volutil (8), bldvldb.sh (8)

AUTHOR

Puneet Kumar, 1990, created

E.29 vrdb(5)

NAME

vrdb - Volume Replication Data Base specification

DESCRIPTION

The volume replication data base is stored in binary form in /vice/db/VRDB on each file server. The *makevrdb* option of the *volutil*(8) program constructs the VRDB on the system control machine (SCM).

The data base consists of fixed-length records, each of which describes a replicated (aka group) volume. Each file server copies the VRDB into memory at start-up and whenever an updated version of it is received. The data base is used to map group volume names and numbers into a VSG and the set of read-write volumes which comprise it.

The VRDB is generated from an ASCII version stored on the SCM in /vice/vol/VRList. The VRList is updated as a side-effect of every create and purge of a replicated volume. Its format is:

<group volname> <group volnum> <# of replicas> <rwvol 1> ... <rwvol 8> <VSG num>

A sample line from the VRList is:

project.coda.src 7f000010 3 c9000012 ca000013 cb000013 0 0 0 0 0 E0000107

Note that all volume and VSG numbers are given in hex. Details of the VRDB structure can be found in <vrdb.h>.

FILES

/vice/db/VRDB

/vice/vol/VRList

BUGS

File servers keep the in-memory copy as a singly-linked list. It should be converted to a pair of hash-tables, one keyed by group volname, the other by group volnum, for fast lookup.

The maximum number of replication sites is fixed at 8. Adding, deleting, or moving replication sites after creation is not supported.

SEE ALSO

volutil (8), maxgroupid (5), vsgdb (5)

AUTHOR

Jay Kistler, 1990, created

E.30 vrlist(5)

NAME

vrlist - vrlist file specification

E.31 vsgdb(5)

NAME

vsgdb - Volume Storage Group Data Base specification

DESCRIPTION

The volume storage group data base is stored in ASCII form in */vice/db/VSGDB* on the system control machine (SCM). It maps VSG numbers into the sets of servers which comprise them. Each VSG is represented by a single line with the following format:

<VSG number> <server-name 1> ... <server-name n>

VSG numbers are in hexadecimal, and must correspond to a legal multicast address as specified in the multicastgroups (5) file. Server names must correspond to entries in the servers (5) data base. There is a limit of 8 on the number of servers in any one VSG. A sample entry from the VSGDB is:

E0000107 mahler vivaldi ravel

The VSGDB is consulted by the *createvol_rep (8)* script when creating a replicated volume. The VSG number specified is looked up in the VSGDB to determine which sites will host the supporting read-write volumes. The VSG number is then wired-into the VRList (and then *vrdb) (5)* entry for the new replicated volume.

FILES

/vice/db/VSGDB

SEE ALSO

servers (5), vrdb (5), createvol_rep (8)

AUTHOR

Jay Kistler, 1990, Created

E.32 vstab(5)

NAME

vstab - vstab file specification

DESCRIPTION

The file */usr/coda/etc/vstab* contains configuration parameters for the Coda cache manager, *venus (8)*. The parameters override default values compiled into Venus, and may themselves be overridden by command-line arguments supplied to Venus at startup.

The file should contain a single line with the following format:

<mount point>:<kernel device>:<default-host list>:<cache directory>:<cache blocks>:<1>

A sample vstab entry is:

/coda:/dev/cfs0:rossini,scarlatti,puccini:/usr/coda/venus.cache:10000:1

Note that the format of the default-host list is <host*1*,host*2*,...,host*n*>. There should be no space between the ":" separator character and the beginning or end of a field.

FILES

/usr/coda/etc/vstab

<*vstab.h*>

BUGS

The last parameter is deprecated and should be removed.

This file is also read by *clog (8)* (and perhaps other programs) which assume that host-list specifies the hosts running auth servers.

SEE ALSO

venus (8), vutil (8)

AUTHOR

Jay Kistler, 1990, Created

E.33 au(8)

NAME

au - add a user

SYNONPSIS

au [-x] [-h host] [-p portal] <cp | cu | nu | du>

DESCRIPTION

au is used to add, delete, or change user information. **au** will first prompt for a current Coda user name and password. If it can authenticate using your user name and password, it will proceed to ask for the new information. **au** supports the following options:

-x

Turns on debugging.

-h *hostname*

Attatch to *hostname* to do the authentication. *hostname* should be the *SCM*.

-p *portal*

Port to bind to. The default portal is coda_auth.

cp

Change password. Use this to change a users vice password.

cu

Change user information. Use this to change password and other user information.

nu

New user. The **nu** option tells **au** to add a new password entry to the vice password database.

du

Delete user. The **du** option tells **au** to delete the given user from the password database.

DIAGNOSTICS

You must be a Coda system administrator to ru **au**.

BUGS

au echos new passwords to the terminal as they are typed in.

AUTHOR

1987, Adapted from AFS-2

Joshua Raiff, 1993, Created man page

E.34 auth2(8)

NAME

auth2 - authentication server

SYNOPSIS

auth2 [-r] [-chk] [-x debuglevel] [-p pwfile] [-tk tokenkey] [-fk filekey]

DESCRIPTION

auth2 is the Coda authentication server. Clients need to have an authentication token from **auth2** in order to establish authenticated connections to Coda file servers.

auth2 supports the following commands line options:

-r

Tells **auth2** to print log entries to the controlling tty rather than a log file.

-chk

Causes **auth2** to only validate passwords. New password entries cannot be added via this server instance. Servers started with the **-chk** option are slave servers. There should only be one master server in the system.

-x ***debuglevel***

Sets the server debug level. This level controls the amount of debugging information printed by the server.

-p ***pwfile***

Used to tell **auth2** where its password file is. Typically this file is /vice/db/auth2.pw.

-tk ***tokenkey***

Specifies where the token key file **auth2** should use when encrypting tokens.

-fk ***filekey***

Name of file containing the key decrypt password file with.

SEE ALSO

authmon (8)

AUTHOR

1987, Adapted from AFS-2.

Joshua Raiff, 1993, Created man page.

E.35 backup(8)

NAME

backup - Volume by volume backup of the Coda File System

SYNOPSIS

backup [-p poll_period] [-t timeout] <dumplist> [dumpdir]

DESCRIPTION

backup performs the clone and dump phases of the Coda backup mechanism. *dumplist* is a file as described in *dumplist*(5). It also reads *vicetab* which is described in *vicetab*(5) to know where to place dump files.

The backup program creates many lines of information as the phases progress. It is a good idea to redirect standard output to a log file. A sample of this log file *backuplogs*(5). After both phases are completed, it prints out a list of volumes in several groupings, and some histograms detailing size and speed of the dumpfiles transferred. The first group are the volumes that were successfully backed up on all servers in their VSG. The second group contains volumes that were successful on some, but not all of their VSGs. The third group contains volumes that were complete failures. The last group contains volumes that are in the VLDB or VRDB but not in the *dumplist*.

The second and third groups use an n-letter word to describe the last successful operation that succeeded on each replica. The kth position in the n-letter word corresponds to the kth replica in the VRDB entry for this volume. One of four letters appears in each position: "L", "C", "D", and "M". "L" means the replica was only locked, "C" means it was cloned but not dumped, "D" means it was dumped (but not marked as such on the server, see the discusion in the manual chapter on backup), and "M" means all phases completed successfully.

backup supports the following command line options:

-p ***poll_period***

Number of seconds to sleep between polls a servers that **backup** thinks are down.

-t ***timeout***

Timout value, in seconds, for RPC2 calls.

SEE ALSO

volutil (8), dumplist (5), backuplogs (5), Backup chapter of the Coda Manual.

AUTHOR

David C. Steere, 1991, Created updated 1998, -hmp

E.36 bldvldb(8)

NAME

bldvldb.sh - build a new Volume Location Data Base (VLDB)

SYNOPSIS

bldvldb.sh

DESCRIPTION

bldvldb.sh builds a new volume location data base **VLDB**. It uses the */vice/db/hosts* file to get a list of servers to use to retrieve the current list of volumes. It gets the list from the */vice/vol/VolumeList* file on each server. It combines the list into */vice/vol/BigVolumeList* and passes the combined list as a parameter to the **volutil makevldb** command. This command produces a new **VLDB** in */vice/db* and updates the files *AllVolumes* and *partitions* in */vice/vol*. You must have root priveledges to run this command.

DIAGNOSTICS

This command can only be issued on the System Control Machine (SCM). Bldvldb uses two mechanisms to get the VolumeList files from the various servers in */vice/db/hosts*. The first is ftp. In order for this to succeed, there has to be a file called ".anonr" in the directory */vice/vol*, which is globally readable and contains a line with the word "VolumeList" in it. If **bldvldb.sh** is unable to get the file via ftp, it will attempt to use the CMU RFS remote file system. If neither mechanism works, **bldvldb.sh** will skip over that servers volumes when building the **VLDB**. The databases are propagated via the update protocol. This takes up to five minutes. Attempts to access a new volume from a client, prior to the propagation of all the databases to all servers, may fail.

FILES

/vice/vol/AllVolumes
is created

/vice/db/VLDB
is created

/vice/vol/VolumeList
on each server is used to build the VLDB

/vice/db/hosts
is a list of the active servers

/vice/vol/BigVolumeList
is the result of combining */vice/vol/VolumeList* from each server

SEE ALSO

update (8), volutil (8)

AUTHOR

Puneet Kumar, 1990, Created

E.37 crvol(8)

NAME

createvol - create a non-replicated volume

SYNOPSIS

createvol <volume-name> <server-name> <partition-name>

DESCRIPTION

createvol is used to create a Coda non-replicated volume on the specified partition (*<partion-name>)* and server (*<server-name>)*. The volume number is assigned by the server automatically; the name is specified by the invoker (*<volume-name>)*.

createvol first checks in */vice/vol/AllVolumes* to see if the volume-name already exists. If not, it uses the *volutil create* command to create the volume at the specified server. It then uses *bldvldb.sh (8)* to build the Volume Location Data Base (VLDB).

The default access control list of the volume gives the group System:Anyuser read and lookup rights. The owner of the root directory, as well as a more reasonable access list, must be set later by a system administrator using the *cfs*(1) command.

EXAMPLE

To Create a volume for user "hbovik" on server "mahler":

```
createvol user.hbovik mahler /vicepa
```

NOTES

This command does not create the mount point to the volume within the file system hierarchy. This must be created by the **cfs mkmount**(1) command.

DIAGNOSTICS

This command must be issued at the System Control Machine (SCM). Also, it must be invoked with effective user id of root. This command does not check for return codes from the servers once the volutil command is made. The invoker must check by hand in the */vice/vol/VolumeList* file on the server to see if the volume name exists at that server.

FILES

/vice/vol/AllVolumes

used to find out if volume already exists

/vice/vol/VolumeList

name of volume created is appended to it

SEE ALSO

cfs (1), bldvldb.sh (8), volutil (8)

AUTHOR

Puneet Kumar, 1990, created

E.38 crvolrep(8)

NAME

createvol_rep - create read-write replicated volume

SYNOPSIS

createvol_rep <volume-name> <vsgaddr> <partition-name> [rep group id]

DESCRIPTION

createvol_rep is a front end to *volutil createvol_rep* and is used to create a Coda read/write replicated volume. The invoker must specify the volume name (*<volume-name>)*, the replication sites via a Volume Server Group address (*<vsgaddr>)*, the partition on which the volume should be created (*<partition-name>)*, and an optional *Group Id*.

createvol_rep first checks in */vice/vol/AllVolumes* and */vice/vol/VRList* to see if the volume name already exists. If not, it uses the *volutil create_rep* command to create the volume at each of the replication sites. It then builds the Volume Location Data Base (VLDB) and the Volume Replication Data Base (VRDB).

The replication sites are determined from the *<vsgaddr>* and the */vice/db/VSGDB* file. The *Rep Group Id* specifies the "replicated" volumeid of the volume being created. By default, the group id in */vice/vol/maxgroupid* is used. Each time it is used it is also updated by adding 1 to it.

After the replicas are created at each replication site, a new VLDB is built automatically using *bldvldb.sh*(8), and the Volume Replication List in */vice/vol/VRList* is updated. The VRList contains one line for each replicated volume. Each line specifies the replicated volume name, group id, number of replication sites, local volume id at each replication site and the VSG address. This file is now used to create a new Volume Replication Data Base (VRDB) using the *"volutil makevrdb /vice/vol/VRList"* command.

EXAMPLES

To create a replicated volume "coda.rep" on 3 sites foo, bar and gorp use:

createvol_rep coda.rep E0000107 /vicapa

where */vice/db/VSGDB* has an entry "E0000107 foo bar gorp" in it.

To assign a predetermined Group Id, use

createvol_rep coda.rep E0000107 /vicepa 7F000003

where "7F000003" is the Group Id.

DIAGNOSTICS

This command must be issued at the System Control Machine (SCM). Also, it must be invoked with effective user id of root. This command does not check for return codes from the *volutil create_rep* command. The invoker must check */vice/vol/VRList* and */vice/vol/VolumeList* at each replication site to see if the volume was created.

FILES

/vice/vol/VRList

contains information on replicated volumes

/vice/vol/VolumeList

name of volume created at each site is appended to it

/vice/db/VRDB

is used to describe the replicated volumes in terms of its non-replicated members.

/vice/db/VSGDB

is used to translate the VSG address to replication sites

/vice/vol/AllVolumes

is used to check if volume exists

/vice/vol/maxgroupid

is used to assign a group id to the replicated volume

SEE ALSO

bldvldb.sh (8), volutil (8), createvol (8)

AUTHOR

Puneet Kumar, 1990, Created

E.39 initpw(8)

NAME

initpw - Initialize the auth2 password database.

SYNOPSIS

initpw [-k key]

DESCRIPTION

initpw initializes the *auth2* password database by reading entries as specified in passwd_coda(5) and writes the corresponding encrypted entries to stdout. **initpw** supports the following command line options:

***-k** key*

Using the **-k** option allows you to specify the encryption key to use when encrypting cleartext passwords. Currently you must give "drseuss " as the key.

initpw expects to receive lines of the form:

```
<ViceId> <Clear Password> <Other user info>
```

See passwd.coda(5).

EXAMPLE

```
initpw -k "drseuss " < /vice/db/passwd.coda > /vice/db/auth2.pw
```

SEE ALSO

auth2 (8), passwd.coda(5)

AUTHOR

1987, Adapted from AFS-2

Joshua Raiff, 1993, Created man page

Lesa Gresh, 1998, modified

E.40 makeftree(8)

NAME

makeftree - format a Coda Data Storage partition to use b-tree structure.

SYNOPSIS

makeftree <vicetab;gt; <partition_path>

DESCRIPTION

`makeftree` organizes an existing UNIX directory with a mounted partition for use by Coda listed in vicetab with an ftree format for fast and efficent file retrival. The ftree infromation is kepe in the file *FTREEDB* at the head of the partition.

The depth and width must be specifed in `vicetab`

BUGS

`makeftree` expects the file FTREEDB to already exist before it is run. Before running

```
touch /<partition_path>/FTREEDB.
```

must be issued to avoid an abort.

SEE ALSO

vicetab (5)

AUTHOR

Henry M. Pierce, 1998, Created man page

E.41 merge(8)

NAME

merge - merge incremental dumps onto full dumps for restore

SYNOPSIS

merge <output file> <full dump> <incremental dump>

DESCRIPTION

merge is a utility that allows one to apply incremental dumps to a full dump and produce a new full dump which reflects a later state of the volume. The new dump can then be used to restore the volume, or can be merged with other incrementals.

An incremental or full dump reflects the state of the volume at the time it was dumped. Full dumps can be restored so that a user may access an older state of a volume. Incremental dumps do not necessarily have sufficient information to be restored. For instance, all the vnodes in the dump may not be present. The merging process allows a full image of a state that was only incrementally dumped to be restored.

Currently incrementals apply to the last successful full dump that was done. As an example, say full dumps for a volume are done on Sundays, with incrementals being taken the rest of the week. If an administrator wishes to restore a volume to Wednesdays state, he would have to fetch the full dumpfile from Sunday and the incremental dumpfile for Wednesday. Once these are present, the administrator would use the merge program to create a new updated full dump for Wednesdays state, and restore it to the server (using volutil(8) restore).

Information in the dump header is used to place a total ordering on the dumps. This way incrementals can only be applied to the dump with repect to which they were taken. In addition, incrementals cannot be applied to other incrementals, and the dumps to be merged must have been created from the same volume replica. The merge program checks these things, and reports failures if they are violated.

SEE ALSO

volutil (8), restore (1)

AUTHOR

David C. Steere, 1991, created

E.42 norton(8)

NAME

norton - Coda file server RVM debugger

SYNOPSIS

Norton

DESCRIPTION

Norton is a utility program that allows you to display Coda file server structures that are stored in *RVM*. Eventually, *Norton* will be a full special purpose debugger that allows you to manipulate the structures as well. *Norton*s command interface uses the gnu *readline* library which features Emacs style command editting as well as maintaining a command history. Command completion is also supported by using <*tab*> and <*esc*>*-?].*

The available commands are:

show directory <*volid*> <*vnum*> <*unique*>

Lists the contents of the directory indicated by *volid.vnum.unique*. Each entries vnode number and uniquifier are also listed.

delete name <*volid*> <*vnode*> <*unique*> <*file*> <*flag*>

remove *file* from the directory specified by *volid.vnode.unique*. If flag is nonzero the linkcount of the directory is reduced by one. **NOTE: delete name does not do anything to the vnodes associated with *file*, you must remove the vnodes by hand or update their link count.**

delete volume <*volid*>

mark a volume with a "destroyme" flag, so that the salvager will destroy it on the next serverstartup.

examine <*addr*> <*len*>

Print <*len*> bytes starting from <*addr*> in hex and ascii.

list volumes

Display a list of all the volumes on the server. This list includes the volume index, name, number, and type.

show free *large* | *small*

Display all of the vnodes on either the large or small free vnode list.

show heap

Display RVM heap usage.

show index <***volname***> | <***volid***>

Display the RVM index of the given volume name or number.

show vnode <***volid***> ***[*** <***vnode***> | ***?*** <**unique**>]

Show the specified vnode. If a *?* is given rather than a vnode number, all of the volumes vnode lists are searched for a vnode with a matching uniquifier.

show volume <***volname***> | <***volid***>

Show a summary of the specified volume.

show volume details <***volname***> | <***volid***>

Show all of the RVM state of the specified volume.

create name < ***parentvolid***>< ***parentvnode***>< ***parentuniqifier***>< ***name***>< ***childvnode***>< ***childuniqifier***>

Insert a vnode in a directory. The parent fid gives the directory in which name is to be inserted. The child fid refers to the child vnode. The link count of the directory is increased if the child is a directory vnode.

x <***addr***> <***len***>

Print <*len*> bytes starting from <*addr*> in hex and ascii. An alias for **examine**.

AUTHOR

Joshua Raiff, 1995, Created. Peter Braam, 1997, new features.

E.43 pcfgen(8)

E.44 prgvol(8)

NAME

purgevol - delete a non-replicated volume

SYNOPSIS

purgevol <volumename>

DESCRIPTION

purgevol is a front-end to the *volutil purgevol* command and is used to delete a non-replicated volume from the Coda system. **purgevol** determines the location of the volume using the readable version of the *VLDB* located in */vice/vol/AllVolumes*. It then uses the *volutil* utility to purge the volume at the volumes site. Finally, it builds a new *VLDB* using the *bldvldb*(8) script.

Purgevol is similar to the *volutil purge*, except that it accepts different parameters, can only be run on the SCM, and builds a new vldb. The user must be root to execute this script.

FILES

/vice/vol/AllVolumes

updated by deleting the entry for the purged volume.

/vice/vol/VolumeList

updated by the remote server as a side effect of purging the volume.

SEE ALSO

bldvldb.sh(8), purgevol_rep(8), volutil(8)

AUTHOR

David Steere, 1990, Created

E.45 prgvolre(8)

NAME

purgevol_rep - delete a replicated volume

SYNOPSIS

purgevol_rep <volumename>

DESCRIPTION

purgevol_rep is a front-end to the *volutil purgevol_rep* command and is used to delete a replicated volume from the Coda system. **purgevol_rep** determines the replicated volumeIds corresponding to this volume by examining the readable version of the *VRDB* located in */vice/vol/VRList*. It then uses the *volutil* utility to purge the individual replicas at the sites of replication. Next, it removes the entry for the deleted volume from the */vice/vol/VRList* and builds a new *VRDB*. Finally, it builds a new *VLDB* using the *bldvldb*(8) script. Like the other volume utilities, **purgevol_rep** must be run on the SCM by root.

FILES

/vice/vol/AllVolumes

updated by removing the entries for the replicas of the purged volume.

/vice/vol/VolumeList

updated by the remote server as a side effect of purging the volume.

/vice/vol/VRList

updated by deleting the entry for the purged volume.

SEE ALSO

bldvldb.sh(8), purgevol(8), volutil(8)

AUTHOR

David Steere, 1990, Created

E.46 pwd2pdb(8)

E.47 readdump(8)

NAME

readdump - Utility to examine dumpfiles created by the Coda backup facility

SYNOPSIS

readdump [dumpfile]

DESCRIPTION

The **readdump** program is an interactive facility to allow the user to get information from a dump file produced by the Coda backup mechanism. For now it is simply a way of looking at the information in a dump file. One cannot use it to modify the dump file.

The **readdump** program supports the following commands:

openDumpFile <dumpfile>

showHeader

showVolumeDiskData

setIndex <index type>

nextVnode

skipVnodes <nVnodes>

quit

readdump uses the *ci* command interface, which allows the use of unique prefixes for commands. Unspecified parameters will be prompted for, and default values can be used. The *showHeader* command prints out the header of the dump, which is seperate from the header of the volume that was dumped. The *showVolumeDiskData* prints out the header of the volume.

The dump contains two streams of vnodes, one for directories and one for files and symbolic links. The *setIndex* command specifies to **readdump** which stream you wish to examine. Once an index has been specified, the *nextVnode* command displays the next Vnode (object) in the stream. Note that movement through the stream is one-directional. To revisit a Vnode, use the *setIndex* command to reset the program back to the beginning of the stream. One can jump through the index by use of the *skipVnodes* command, which takes the number of vnodes to skip, and reads past (without displaying) that many vnodes in the stream.

Future Work

Usage of this facility should suggest more commands to help administrators parse dumpfiles. In particular, a command to seek ahead to a specified offset in the dump would be useful.

SEE ALSO

merge (8), backup (8), volutil (8), Coda Manual

AUTHOR

David C. Steere, 1991, Created

E.48 codasrv(8)

NAME

codasrv - CFS file server

SYNOPSIS

codasrv [-d (debug level)] [-rpcdebug rpc2debuglevel] [-p (number of processes)] [-b (buffers)]

[-l (large vnodes)] [-s (small vnodes)] [-k (stack size)]

[-w (wait interval)] [-r (RPC retry count)]

[-o (RPC timeout)] [-c (check interval)]

[-t (RPC trace buffers)] [-forcesalvage]

[-quicksalvage] [-rvm logdevice datadevice length] [-nores]

[-trunc percent] [-nocmp] [-nopy] [-nodumpvm] [-nosalvageonshutdown]

[-mondhost hostname] [-mondportal port] [-debarrenize] [-optstore]

[-rvmopt] [-newchecklevel checklevel]

DESCRIPTION

codasrv is the CFS *vice* file server. It services requests from client machines **venus** processes and maintains the file system.

Srvs command line options are:

-d <***debug level***>

Sets the internal debugging level to <*debug level*>. This controls the amount of debugging output **codasrv** will generate.

-p <***number of processes***>

Sets the number of light weight processes that will be used to handle concurrent requests.

-b ***buffers***

Sets the number of internal buffers used by **codasrv** for the dir package.

-l <***large vnodes***>

number of large vnodes in lru cache

-s <***small vnodes***>

number of small vnodes in lru cache

-k <***stack size***>

stack size of LWP threads in Kbytes

-w <***wait interval***>

interval for CallBackCheckLWP to see which clients are alive

-r <***RPC retry count***>

number of times a call is retried before reporting death

-o <***RPC timeout***>

default timeout for all rpcs

-c <***check interval***>

The interval at which shutdown is checked

-t <***RPC trace buffers***>

number of entries in the rpc trace buffer

-forcesalvage

force a full salvage on all volumes

-quicksalvage

salvage only the headers of the volumes - currently it always does a full

-rvm ***logdevice datadevice length***

The **-rvm** switch allows you to specify the locaction of the log device, data device, and length of the recoverable segment used by the server. This parameter is not optional.

-nores

Turn off resolution on the servers

-trunc ***percent***

Specify how full the log can get before it is truncated. Default is 50%

-nocmp

Directory replica contents will not be checked for equality at end of resolution

-nopy

Suppress polling and yielding by threads that run for a long period

-nodumpvm

Do not dump in-memory copy of recoverable segment before shutdown

-nosalvageonshutdown

Do not salvage volumes on shutdown

-mondhost *hostname*

Host where *mond* database is stored

-mondportal *port*

Port to use when contacting the *mond* host.

-debarrenize

Create an inode for a vnode if the inode cannot be found by the salvager

-opstore

Optimized Stores - return to the client before sftp completes

-rvmopt

Turn on inter/intra transaction rvm optimizations

-newchecklevel *checklevel*

Set the level of checking done by new plumber.

FILES

/vice/srv/SrvErr

/vice/srv/SrvLog*

SEE ALSO

startserver (8)

AUTHOR

Joshua Raiff, 1993, Created man page

E.49 startserver(8)

NAME

startserver - Start the CFS file server

SYNOPSIS

startserver

DESCRIPTION

startserver is used to start the CFS file server process. It will clean up old log files, then start the server. This script is useful for starting the server with the same configuration every time its run.

FILES

codasrv

/vice/srv/SrvErr

*/vice/srv/SrvLog**

/vice/srv/srv.conf

SEE ALSO

codasrv (8)

AUTHOR

Joshua Raiff, 1993, Created man page

E.50 updtclnt(8)

NAME

updateclnt - update client executables

SYNOPSIS

updateclnt [-d debug_level] [-h host_name] [-q service_name] [-w wait_interval] [-r reps]

DESCRIPTION

The **updateclnt** command is a client process to the *updatesrv* process. It is used to keep the binaries and data bases on a file server in sync with those on a control machine. The arguments include:

-d ***debug_level***

This option changes the debugging level. The higher the debug_level, the more information is printed. Maximum debugging level is 10.

-h ***host_name***

This is the hostname of the control machine with which this process keeps up-to-date. Default: scm.

-q ***service_name***

This is the name in */etc/services* that should be used as the connect portal on the machine specified by **host_name** above. Default: rupdsrv.

-w ***wait_interval***

The interval between probes, in seconds. The lower this number, the quicker the servers will be updated when a change occurs and the more cpu and network resources required. Default: 300s (5min).

-b

This option forces a .BAK file to be kept for any file that is changed.

-r ***reps***

This is the number of wait intervals between long checks. Files are checked at two intervals. The interval specified by **-w** is used for those files in */vice/db*. All other files are checked once each **-r** repetitions of length **-w**. Default: 6, so the long interval is 30min.

Updateclnt checks */vice/db* every *wait_interval* seconds. This directory contains a file called *files*. Each file specified in it has its current date checked at the server, and a new copy of the file is fetched if the date does not match that on the control machine. The format of the *files* file is defined below.

EXAMPLES

The command **updateclnt -h mahler** would cause the client machine to check the host "mahler" every 5 minutes to see if any of the */vice/db* files have changed and every 30 minutes to see if any other files have changed. Normally the command is invoked by issuing the **updatemon (8)** command with the same operands.

FORMAT

The format of the *files* file is one file name per line. A "-" as the first character on the line causes the file to be deleted instead of copied.

DIAGNOSTICS

The updateclnt program can have its debug level turned on while it is running by sending a **kill -TSTP** signal to a running updateclnt. To reset the debug level back to zero, send a **kill -HUP** signal to the running updateclnt. This also causes the UpdateLog file to not be written anymore.

FILES

/vice/srv/UpdateLog

is the name of the log file for the updateclnt command. It can be viewed by using the command *tail -f /vice/file/UpdateLog*. It is only written if debugging has been started.

/vice/srv/UpdatePid

is the pid of the updateclnt process.

/vice/srv/UpdateMon

is the pid of the updatemon process that is keeping updateclnt running.

BUGS

There is no easy way to add to the list of directories checked.

The *-r* option is now obsolete as **updateclnt** does not update any directory other then */vice/db*.

SEE ALSO

updatesrv (8)

AUTHOR

Maria R. Ebling, 1990, Adapted from AFS

E.51 updtsrv(8)

NAME

updatesrv - update server files

SYNOPSIS

updatesrv [**-d** debug_level] [**-l** number_of_lwps] [**-p** prefix] [**-s**] [**-q** service_number]

DESCRIPTION

The **updatesrv** command is a read-only server that responds to requests for files by name. The operands that it accepts are:

-d debug_level

This is used to increase the level of messages written to the log file.

-l *number_ of_ lwps*

This operand is used to specify how many lwps will be used to answer file requests. Default: 2.

-p *prefix*

This name, prefixed to all file requests, is used to limit the files that can be requested.

-q *service_ name*

This is the service name in */etc/services* that is to be used. Default: rupdsrv.

EXAMPLES

The command **updatesrv -s -p /** would invoke the updatesrv process and cause the files to be served relative to / on the server machine.

The command **updatesrv -s -p /vice/bin** would cause the files to be served relative to /vice/bin on the server machine.

DIAGNOSTICS

The UpdateLog file will contain useful information. Issuing a **kill -TSTP** signal to a running updatesrv will increase the level of debugging messages. The debugging messages can be turned off by issuing a **kill -HUP** signal. This will also cause the log file to no longer be written.

FILES

UpdateLog

This log of the messages from **updatesrv** is kept in the directory specified by the **-p** operand on the invocation. This log file is only written when debugging is enabled.

pid

This file contains the pid of the updatesrv process. It is kept in the directory specified on the **-p** operand on invocation of the command.

BUGS

It does not yet use any password authentication to stop others from using it to fetch files from servers.

SEE ALSO

checkvenus(8) updateclnt(8)

AUTHOR

Maria R. Ebling, 1990, Adapted from AFS-2

E.52 venus(8)

NAME

venus - client cache manager

SYNOPSIS

venus [-k kernel device] [-h host list] [-cf cache files] [-c cache blocks] [-mles CML entries]

[-d debug level] [-rpcdebug rpc2debuglevel] [-r root volume] [-f cache directory] [-m COP modes]

[-mc 0 | 1] [-console console file] [-retries RPC2 retries] [-timeout RPC2 timeout]

[-ws SFTP window size] [-sa SFTP send ahead] [-ap SFTP ack point] [-init] [-p]

[-hdbes hoard entries] [-rvmt RVM type] [-sim infile outfile filtfile systype]

[-mondhost hostname] [-mondport port] [-maxprefetchers fetch threads]

[-maxworkers work threads] [-maxcbservers cb threads] [-vld device] [-vlds size]

[-vdd device] [-vdds size] [-rdscs chunk size] [-rdsnl nlists] [-logopts 0 | 1]

[-swt weight] [-mwt weight] [-ssf scale factor] [mount point]

DESCRIPTION

venus manages a cache of files and directories for a client workstation. It has a host of parameters and configuration options. Default values of everything are compiled into **venus** (see <*venus.private.h*>*) for these values*. Some of these are overridden by the values in the *vstab*(5), provided that it exists. Both default and vstab values may be overridden with command-line arguments. Venus must be run as root.

The command-line options are:

-k *kernel device*

Take *kernel device* to be the device for kernel/venus communication.

Default: */dev/cfs0*

-h *host list*

Take comma-separated *host list* to be the initial set of hosts to be used for volume-location requests.

-cf *cache files*

Limit the size of the file cache to *cache files* entries.

Default: *512*

-c *cache blocks*

Limit the size of the file cache to *cache blocks* 1K blocks.

Default: *8192*

-mles ***CML entries***

Number of Cache Modify Log entries

-vols ***volumes***

Limit the size of the volume cache to *volumes* entries.

Default: *128*

-vsgs ***vsgs***

Limit the size of the VSG cache to *vsgs* entries.

Default: *64*

-d ***debug level***

Initialize the debug level to *debug level*. Meaningful values are {0, 1, 10, 100, 1000}.

Default: *0*

-r ***root volume***

Take *root volume* to be the root volume of the file system. This should be a string name rather than a number. There is no default. If no value is given, venus will ask one of the file servers. [N.B. If you intend to operate disconnected, a value must be supplied.]

Default: *none*

-f ***cache directory***

Take *cache directory* to be the directory for the file, volume, and VSG caches. Venus garbage collects any files it doesnt recognize in the cache directory, so use caution when supplying this argument. Venus will create the directory if it doesnt already exist. The directory should have mode bits of *rwx*—— to protect the cache from malicious local users.

Default: */usr/coda/venus.cache*

-m ***COP modes***

Controls what Coda Optimistic Protocol (COP) options are enabled. *COP modes* is interpreted according to the following bit-mask: [PIGGYCOP2 | ASYNCCOP2 | ASYNCCOP1]. Only some combinations are legal.

Default: *[PIGGYCOP2 | ASYNCCOP2]*

-mc ***0*** | ***1***

Controls whether multicast is used or not (0

-console ***console file***

Redirects console messages to *console file*.

Default: */dev/console*

-retries *RPC2 retries*

Sets the number of RPC2 retries to *RPC2 retries*.

Default: *4*

-timeout *RPC2 timeout*

Sets the RPC2 timeout period to *RPC2 timeout* seconds.

Default: *30*

-ws *SFTP window size*

Sets the SFTP window size to *SFTP window size* packets.

Default: *8*

-sa *SFTP send ahead*

Sets the SFTP send ahead to *SFTP send ahead* packets.

Default: *4*

-ap *SFTP ack point*

Sets the SFTP ack point to *SFTP ack point* packets.

Default: *4*

-init

Initializes LEFTPAREN i.e., clears) file, volume, and VSG caches.

-p

Enables profiling.

-hdbes *hoard entries*

Number of hoard database entries.

-rvmt *RVMT type*

Media that RVM resides on. Meaningful values are: 1

-sim *infile outfile filtfile systype*

Turn on simulator mode. *infile* is a trace to load in to the simulation. *outfile* is the output file for the simulation. *filtfile* is a filter for the trace. And *systype* is the system type the trace was taken on.

-mondhost *hostname*

Host where *mond* database is stored

-mondportal *port*

Port to use when contacting the *mond* host.

-maxprefetchers *fetch threads*

Maximum number of threads doing prefetch ioctls

-maxworkers *work threads*

Number of worker threads.

-maxcbservers *cb threads*

Number of callback server threads.

-vld *device*

Location of the venus log device.

Default: */usr/coda/LOG*

-vlds *size*

Size of the log device.

-vdd *device*

Location of the venus data device.

Default: */usr/coda/DATA*

-rdscs *chunk size*

Specify RDS chunk size.

-rdsnl *nlists*

Number of RDS nlists.

-logopts *0* | *1*

Turn on log optimization.

-swt *weight*

Short term cache priority weight.

-mwt *weight*

Medium term cache priority weight.

-ssf *scale factor*

Short term cache scale factor.

mount point

Directory to serve as the mount point for Coda.

Default: */coda*

DIAGNOSTICS

Venus writes debugging information into the file <cache directory>/venus.log. The verbosity of this output is controlled by the <debug level> parameter. High priority messages are also written to the console (which may be redirected with the console option at start-up). Fatal errors will cause the internal state of venus to be dumped to the log file, and a core file to be left in <cache directory/core>.

Venus writes its process id into the file <cache directory>/pid. The vutil (8v) program reads the pid file and dynamically alter Venus behavior by sending signals to it.

Venus may be unable to unmount itself cleanly when it exits. Usually this is due to processes which have references to vnodes in the Coda namespace (e.g., a process is cded somewhere in Coda). Once these references are released, Venus can be unmounted with cfsunmount (8v).

FILES

/usr/coda/etc/vstab

<cache directory>/venus.log

<cache directory>/pid

<venus.private.h>

BUGS

A mount point of */coda* is assumed by the pioctl library.

A cache directory of */usr/coda/venus.cache* is assumed by vutil (8v).

SEE ALSO

vstab (5), cfsunmount (8v), vutil (8v)

AUTHOR

Jay Kistler, 1990, Created

Joshua Raiff, 1993, Documented added switches

E.53 volutil(8)

NAME

volutil - volume utility subsystem

SYNOPSIS

volutil [-h <server>] [-t timeout] [-d debuglevel] <command> <parameters ... >

DESCRIPTION

Volutil is a RPC interface to the volume utility subsystem of the file server. The volume utilities are used to perform administrative functions like creating, cloning, purging, dumping and restoring volumes. Each of these functions can be invoked via the <**command**> parameter. Each command has a set of parameters that it expects. These are listed below along with a short description of each command. The volutil utility may be instructed to perform the operations on a server at a remote site by specifying the server to which to connect with the -h option. The default is to connect to a server on the local machine. The -t option may be used to specify the timeout (in seconds) to be used by RPC2.

ancient <***group-ID***> <***rep-ID***>

Tell the server that backup succeeded for this volume. The next dump of this volume, if incremental, will be based on the state represented by this backup. The input should be in Hex.

backup <***volume-ID***>

Create a backup clone of a read/write volume. If a backup clone already exists, update it to reflect the current state of the read/write volume; Otherwise, create a new read-only volume. The read/write volume must be locked for this to succeed. Backup unlocks the volume as a side effect.

clone ***volume-ID*** **[-n** ***new-volume-name***]

Create a read only clone of a read write volume with (replica) ID (<*volume-ID*>). The vnodes are actually copied but the inodes are marked copy-on-write i.e. inodes need to be copied only if modified. The name of the new cloned volume can be optionally specified by the <*new-volume-name*> parameter. Default value is volume-name.readonly. The *clone*(8) command can be used to call **volutil clone**.

create <***partition-path***> <***volume-name***>

Create a nonreplicated read-write volume named <*volume-name*> on partition named <*partition-path*>. The *createvol*(8) command is a simplified front-end for this option.

create_rep <***partition-path***> <***volume-name***> <***group-ID***>

Create a replicated read-write volume named <*volume-name*> on partition named <*partition-path*>. The <*group-ID*> parameter is used to specify the ID of the replicated volume to which this replica will belong. The *createvol_rep*(8) command will also create a replicated volume.

dump [-i *<volume-ID> <file-name>*]

Dump the entire contents of a volume (*volume-ID* in Hex) to a file (*file-name*). If the -i flag is used, the dump will be incremental, it will only include vnodes which have been modified since the last dump was taken. The dump is not machine independent, certain fields in the Vnode are not translated to network order. However, dump files can be used to create restored volumes on machines with a similar byte-order.

dumpmem ***<file-name> <address> <size>***

Dump <*size*> bytes starting at <*address*> into <*file-name*>.

elapse ***<on | off> <resolution | cb | mond | volDump> [MultiRPC]***

Turn on or off timers for the given subsystem.

info [-all *<volume-name | volume-number> <file-name>*]

Print in ascii the contents of a volume into a file (*<file-name>)*. The volume can be specified by its name, or by the volume-ID, specified in Hex. If -all is specified, contents of both large and small vnodes in that volume are also printed.

lock ***<volume-ID>***

Lock a volume to prevent write access to the volume. To other users it will appear as if the operation would time out.

lookup ***<volume-name | volume-ID> <file-name>***

Print information about a volume (specified by *volume-name* or *volume-ID* specified in Hex) in *file-name*. Only meta-information, such as replicated group-ID, location, etc. is printed.

makevldb ***<VolumeList>***

Create a new Volume Location Database **VLDB**. *VolumeList* names a file containing volume parameters for all volumes in the system. This command typically is run on the system control machine. See also *bldvldb*(8) and *volumelist*(5).

makevrdb ***<vrlist>***

Create a new Volume Replication Data Base (**VRDB**). <*vrlist*> is a file containing entries describing replicated volumes. Each entry contains the name, group-ID, read-write ids, and the VSG address of a replicated volume. There is one entry per replicated volume in the system.

purge ***<volume-ID> <volume-name>***

Delete a volume. For replicated volumes **purge** must be called from each server individually on the replicas at the different servers. (See purgevol-rep(8))

restore ***<file-name> <partition-path> [<volume-ID> <volume-name>]***

Create a new volume on the partition named by <*partition-path*> and read in the contents from a dump in file <*file-name*>. The new volume will be given the name and ID specified on the command line. If either is unspecified, or if the Volume ID is of illegal form, the server will allocate the ID or name based on internal rules. The volume-ID should be specified in Hex.

rvmsize *<volume-ID>*

Print the RVM statistics for the volume *<volume-ID>*.

setdebug *<level>*

Set the debug level for the volume and directory packages to *level*. To reset it use zero for the *level* argument. The rpc2 debug level is set to level/10.

setlogparms *<volume-ID>* **[<*reson*> <1 | 0>** [<*logsize*> <*size*>]]

Turn on resolution or change the log size for a volume. The volume ID can be either the replicated ID or the non-replicated ID. Resolution is turned on by specifying 4 after *reson* and can be turned off by specifying 0. The size of the log can also be changed for the volume. The *size* parameter refers to the number of maximum entries in the log. This should be a multiple of 32. Typically this is set to 8192.

setvv *<volumeNumber> <vnodeNumber> <Uniquifier> <version numbers (8)> <host unique flags>*

Set the version vector for a vnode (with fid = <*volumeNumber*>.<*vnodeNumber*>.<*Uniquifier*>). The new version vector is specified by the <*version numbers*> and the <*host unique flags*> triple.

showvnode *<volumeNumber> <vnodeNumber> <Uniquifier> <file-name>*

Print the contents of a vnode (with fid = <*volumeNumber*>.<*vnodeNumber*>.<*Uniquifier*>). into <*file-name*>.

shutdown

Bring all volumes offline and bring down the server.

swaplog

Save the fileserver log output (in *SrvLog.old*) and start a new SrvLog.

timing *on* | *off* *<filename>*

Turn timing on or off.

truncatervmlog

Forcibly truncate the RVM log.

unlock *<volume-ID>*

Unlock a volume locked via the **volutil** lock command. (*volume-ID) should be in Hex*

updatedb

Make the file server read in a new VLDB and VRDB. The server assumes the databases to exist in */vice/db/VLDB* and */vice/db/VRDB*. This utility is useful for reading in new databases at non-scm machines.

DIAGNOSTICS

This command must be run as root. It works only on the machine running the server. Also, the token file */vice/db/voutil.tk* must be accessible.

BUGS

The salvage option to volutil doesn't work right. Please don't try it.

FILES

/vice/db/VSGDB

/vice/file/SrvLog

/vice/db/VLDB

/vice/db/VRDB

/vice/vol/VRList

SEE ALSO

vrdb(5), volumelist(5), bldvldb(8), createvol(8), createvol_rep(8), purgevol(8), purgevol_rep(8)

AUTHOR

Puneet Kumar, David Steere, 1990, Created

E.54 vutil(8)

NAME

vutil - venus utility program

SYNOPSIS

vutil [-cop COP modes] [-cs] [-d debug level] [-mc 0 | 1] [-p 0 | 1]

[-shutdown] [-stats] [-swap]

DESCRIPTION

vutil dynamically alters various parameters on a running Venus. It reads the value of the file /usr/coda/venus.cache/pid and sends signals to that process.

The parameters are:

-cop ***COP modes***

Controls what Coda Optimistic Protocol (COP) options are enabled. *COP modes* is interpreted according to the following bit-mask: [PIGGYCOP2 | ASYNCCOP2 | ASYNCCOP1]. Only some combinations are legal.

-cs

Clears internal counters used in statistics reporting.

-d ***debug level***

Sets the debug level to *debug level*. Meaningful values are {0, 1, 10, 100, 1000}.

-mc ***0*** | ***1***

Enable/disables multicast (0 = “OFF,” 1 = “ON”)

-p ***0*** | ***1***

Enables/disables profiling (0 = “OFF,” 1 = “ON”)

-shutdown

Shuts Venus down gracefully.

-stats

Causes Venus to dump internal counters and other statistics to its log file.

-swap

Causes Venus to move its current log file to <log-file>.old and reinitialize <log-file>.

DIAGNOSTICS

Only the super user can run **vutil**.

FILES

/usr/coda/venus.cache/pid

/usr/coda/venus.cahe/venus.log

BUGS

Venus cache directory is assumed to be */usr/coda/venus.cache*.

SEE ALSO

venus LEFTPAREN 8)

AUTHOR

Jay Kistler, 1990, Created