Kara - A Distributed Configuration Management System for OpenBSD

Adrian Close

Fernhill Technology

<adrian@fernhilltec.com.au>

ABSTRACT

Anyone running computer systems eventually strikes the problem of configuration management. The simplest approach might be to maintain configuration directly on a system, making changes by hand, however recovering from a disaster can be quite difficult if no copies of configuration files exist. Managing multiple systems in this fashion increases the disaster potential significantly.

Clearly, backups are an important aspect of system management. Revision control of configuration files is a useful tool, especially when multiple system administrators and operators are involved. Repeatable methods of configuration change deployment can be critical to service availability. The ability to automate these tasks can make the difference between getting the job done and total work overload, especially if you are looking down the barrel of a task like managing hundreds of systems.

Kara is a framework designed to address these issues. It is not a single bit of software that will make all your problems go away. Rather it is a methodology, combining a number of well-known and freely available software packages (CVS, SSH and others) with system administration techniques gleaned through some years of experience. It still won't make all your problems go away, but it should help you keep important configuration files under control.

Kara is designed not to get in your face unless you want it to. You can manage as much or as little of your configuration as you like (even just one file), so there's no "flag day" requirement.

It is unashamedly *nix-centric, aimed in particular (at least in the first instance) at OpenBSD systems. It should prove easily adapatable to other BSDs and Linux-based operating systems, as well as some commerical flavours. If you're unfortunate enough to be saddled with operating systems that don't support text-based configuration files, Kara is probably not the answer.

1. Kara's Best Friend - CVS

Revision control is probably the single most important aspect of configuration management. In our case, CVS provides the core "database"-like area (known in CVS parlance as the "repository") and functionality needed to keep track of system configuration files, as well as the ability for multiple people to make changes in an orderly fashion.

Normally, CVS is deployed with a single repository and used to manage source code. However, our situation is a little more complicated. In particular:

- The machines we are managing may be scattered across the network, with varying amounts of connectivity and reachability. Storing all of the configuration files in the one CVS repository is not very practical under these circumstances. We don't want a situation where network connectivity is required in order to restore network connectivity.
- We are also interested in the ability to make changes from the machine itself, both for the above reasons and the desire not to rely on centralised infrastructure. In fact, being able to make changes from most anywhere (assuming appropriate security measures) would be a definite boon.
- Given the varying security profiles of systems under management, allowing access from such systems into a central configuration repository is dangerous at best. You might trust users of certain machines to make changes to their own environment, but you might not want to give them full access to everything.
- We need standardised methods of deploying changes to system configuration. Not all changes are a simple matter of copying a file

into '/etc'. Further, methods of applying changes differ between operating systems and indeed, releases of operating systems.

2. Getting Started - CVS on 'localhost'

Sometimes theory is best learnt by practice. Here's a basic recipe for implementing Kara on a host:

- Top level directory '/localhost'. Yes, I can hear the cries of despair from here - "nooooo, not another top level directory." Work with me on this one. It's part of my design for a Grand UniFied [network] Filesystem ('guff' for short). If you don't like it, feel free to put it somewhere else ('/etc/localhost' might make sense), but hear me out before you go off in a huff.
- Underneath, '/localhost/svc/cvs'. This is the CVS repository for this particular system (i.e. "localhost"). You'll need to be sure and set up a file mode/permissions structure that allows your user account read/write access the repository (you can't check stuff in to CVS as 'root', although you can check stuff out). You'll need to create '/localhost/svc/cvs/CVSROOT' too. A good way to do this is with the 'cvs init' command, so you get all the funky CVS control files.
- '/localhost/svc/cvs/kara/config'. This is the CVS "module" into which we'll put the system's configuration files. Underneath this, we want directories 'files/etc'. This should start to look vaguely familiar. (Now, don't ever mess directly with the CVS repository again or the CVS police will hunt you down.[1])
- You may wish to consider file permissions and ownership within the repository, especially if you don't want everyone on the system to be able to make configuration changes (then again, you might).
- You'll want a checkout somewhere of the 'config' module so you can play with it. Set the CVSROOT environment variable to "/localhost/svc/cvs" and run 'cvs get kara/config'. You should get a 'config' directory with subdirectories 'files/etc'. Go there.
- Copy in the files you want to manage from '/etc' and use 'cvs add *' to tell CVS about them. Then do 'cvs commit' to confirm the changes. CVS will ask you for comments. I highly recommend them, even if it's just something like "initial check-in", since they send a clear message about what you were *trying* to do, as opposed to what you actually did (which you can find using 'cvs diff').
- Use a CVS tag on the CVS versions of files you want to make live on the system. We suggest something obvious like "LIVE". This allows you to check files into CVS and let other people look at your proposed changes before they are made live.
- '/localhost/opt/kara/var/staging/localhost/config'. This is a checkout of the aforementioned CVS module which we'll use to stage the configuration files to be made live on the system (cd to '/localhost' and run 'cvs get -rLIVE kara/config' to make this).
  You'll want to update ('cvs up -rLIVE') this on a regular basis and a crontab entry is probably the easiest way.

There are several fine features to this arrangement, not the least of which is that you can have a number of people checking out the configuration database and working on it from a number of locations. CVS can be used over SSH, so you can even do this securely over an insecure Internet.

3. The Missing Link

The astute reader will have noticed that we do not as yet have any link back from the CVS repository to the actual business end of the operating system's live configuration (such as '/etc'). All we have is a checkout of some files under '/localhost/opt/kara/var/staging/localhost/config/files/etc'.

You could do something simple like create symbolic links from '/etc' to the appropriate files in the checkout area. This would be OK in many cases, but sometimes changes in configuration files require special action to take effect (such as HUP'ing a daemon or running a program to import a plaintext file into a database).

Since we're aiming at a repeatable and largely automatable system, we probably want something like a collection of installation shell scripts (for one thing, this mitigates the risk of operator error). And since we're also looking at running this system on a potentially large number of machines, some way of distributing these shell scripts and keeping them up to date is also desirable (preferably without having to

1. There *are* times when performing actions on the CVS repository directly is desirable, or even necessary, such as relocating or removing modules if you make a mistake. Do so with extreme care. Also, note that this sort of modification can cause existing checkouts to break, so you might want to just delete them and check out again. If this seems impractical, you now understand why direct repository operations are evil.

manually install updates on each machine). Ideally the configuration scripts would adhere to the Unix philosophy of "doing one thing and doing it well" - default scripts to handle most cases, specific scripts for special cases and an uber-script to tie them all together.

And so we turn once more to our good friend CVS…

4. A Tree of Configuration Actions

As much as we try to avoid centralisation, some does inevitably creep in. In order to maintain our collection of shell scripts we need somewhere to put them. A CVS repository is a pretty good way of doing this, especially since CVS was designed for managing software projects.

We think that three possible configuration actions per file are enough to handle any situation, namely:

- 'pre.sh' - e.g. shut down a daemon.
- 'install.sh' - e.g. copy the file.
- 'post.sh' - e.g. start the daemon again.

1. First, we need a central host to hold the master repository. This can effectively be anywhere, but your choice of location should be relatively secure (since anyone with access to the repository can have a significant deleterious effect on the operation of your systems) and well connected (since you'll want remote systems to be able to talk to it).

   If you're looking for a little more reliability in your master repository, it's worth noting that the client nodes just need read-only access. You could create multiple read-only repositories, updated from the master.
   In fact, a read-only repository (i.e. a copy of the master) is a fine idea, since the client nodes themselves should never need to update the action scripts. Since it's read-only, there's probably no reason not to make it availble via 'anoncvs' (see http://www.openbsd.org/anoncvs.shar for an example), unless you have other stuff in the repository, which I wouldn't recommend.

2. On the master system you choose, create a CVS repository (see above). You'll want a module called 'kara'. We put ours in '/localhost/opt/kara/svc/cvs'.

   (With a bit of luck, you'll have a distribution with a 'tar' file containing a pre-populated repository.)

3. Since the way things are done often changes subtly between operating system releases, it is probably wise to maintain distinct sets of configuration files per major operating system revision. Create a 'releases' module under 'kara' in the repository and underneath, a module name that makes sense for the operating system revision that you're dealing with (in our case we chose '3.5' since we're only dealing with OpenBSD and have no great need to maintain two sets of version numbers - Kara 3.5 goes with OpenBSD 3.5).

   We've chosen to make one directory per release, rather than use the branching capabilities of CVS, as we feel this path is less prone to confusion come upgrade time. A simple symbolic link to the required version is probably more obvious than CVS arcana…

4. Now we have modules 'kara/releases/3.5'. Underneath this create modules 'config/actions/'. This is where the actual installation scripts will live.

5. Create 'kara/releases/3.5/opt/kara-tools'. This is where our "uber-script" that drives all the others will live.

6. Drilling down still further, create modules 'kara/releases/3.5/config/actions/kara/defaults' and 'kara/releases/3.5/config/actions/etc'.

   - The 'defaults' module holds the default installation scripts. Probably the only sensible default is an 'install.sh' that copies the file in question.
   - The top level of the 'actions' module mirrors the filesystem, containing one directory per file. Each file's directory contains one or more of 'pre.sh', 'install.sh' and 'post.sh', as required for that file. If a script does exist, the "uber-script" uses the default instead.

5. Action Script Examples

The configuration action scripts can (and probably should) be quite simple. Updating system configurations is fraught with danger at the best of times and automation introduces the potential for automated failure. In the face of an error, we'd like the system to a) refrain from making any further changes (thereby hopefully limiting the damage) and b) tell a human.

It is intended that the system be driven by 'cron', which mails the output of any jobs to the controlling user (probably 'root'). If you make sure that mailbox actually gets read, you should get the important news from Kara.

Our standard system expects the scripts to be shell scripts, but there's no real reason you couldn't extend the system to support scripts written in Perl, or Python, or anything else that takes your fancy. Just remember to incorporate lots of error checking in your scripts.

5.1 Example default 'install.sh'

```
#!/bin/sh

echo "Kara default configuration file \
      install script ($2):"

echo -n Copying $1 to $2...

cp $1 $2

if [ $? != 0 ]; then
        echo Error - $0 exiting.
        exit 1
fi

echo done.
```

Notably, as it is, this script does not create directories. So if you add a file in a directory in your CVS confguration checkout, the 'cp' command will fail, the Kara update process will fail and you'll get mail. We think this is a feature - adding new configuration directories is generally a significant event and we like the extra check and balance the default script gives us.

If you don't, it should be easy to extend the script to auto-create directories. Better yet, create a file-specific install script that makes them and leave the default script as a warning.

5.2 Example 'post.sh' (for Squid)

```
#!/bin/sh

echo "/etc/squid/squid.conf post-install \
     script ($2):"

echo -n Restarting squid...

/usr/local/sbin/squid -k reconfigure

if [ $? != 0 ]; then
        echo Error - $0 exiting.
        exit 1
fi

echo done.
```

6. Introducing… The Uber-Script(s)!

So, we're nearly there. We have a CVS repository of configuration files and a CVS repository of action scripts that know what to do with them. We just need something to glue them together - in particular, to work out when/what configuration files changes and which script to run to put them into place. Having identified two distinct tasks, following the Unix philosphy, it seems to make sense to create distinct programs to deal with them separately.

Our first script examines the repository and produces a list of files needing attention, storing this in an intermediate format for use by the second script. It does this by simply running a CVS update on a "staging" checkout of the configuration repository and examining the output.

This separation of function also gives us the useful ability to manually create or tweak the intermediate file to request specific actions manually, say for debugging purposes.

6.1 Example "uber-script" config file

This is currently in the repository as 'kara/release/3.5/opt/kara-tools/kara-config.props'.

```
CONFIGFILECHECKOUT=/localhost/opt/kara/var/
staging/localhost/config/files/
CONFIGACTIONCHECKOUT=/localhost/opt/kara/
libexec/release/3.5/config/actions/
default_installscript=/localhost/opt/kara/
libexec/release/3.5/config/actions/kara/
defaults/install.sh
cvsworkfile=/localhost/opt/kara/var/staging/
localhost/config/cvsworkfile
```

6.2 Example "uber-script" - part 1

This is currently in the repository as 'kara/release/3.5/opt/kara-tools/cvsup-buildprops'.

```
#!/usr/bin/perl

# Required modules
use strict;
use FileHandle;
use CS::Properties();

# Get config
my $configprops = new \
    CS::Properties("kara-config.props");
my $CONFIGFILECHECKOUT= \
    $configprops->get("CONFIGFILECHECKOUT");
my $cvsworkfile=
    $configprops->get("cvsworkfile");

# This is how we run CVS
my @cvscmd = (
   "cd $CONFIGFILECHECKOUT; cvs -q -d update"
);

my $cvsoutputprops = new \
    CS::Properties($cvsworkfile);

# read any pre-existing, pending CVS output
$cvsoutputprops->read($cvsworkfile);

# Run CVS update and build properties list of
# any new files to process.
my $fhcvscmd = new FileHandle;
open $fhcvscmd, "@cvscmd|" or die \
    ("failed to run CVS command\n");
my $line = $fhcvscmd->getline();
while ($line) {
    $line =~ /^(.)\s+(.+)$/;
```

```
    my $cvsAction = $1;
    my $cvsFile = $2;
    my $cvsStatus = "todo-pre";
    my $item = [ $cvsAction, $cvsStatus ];
    $cvsoutputprops->put($cvsFile, $item, \
        time());
    $line = $fhcvscmd->getline();
}
$fhcvscmd->close();

# Flush list of updated files
$cvsoutputprops->write($cvsworkfile);
```

6.3 Sample intermediate file ('cvsworkfile')

The intermediate format is borrowed somewhat from the land of Java, and is basically name/value pairs in a text file, extended to store arrays. A fine man by the name of Craig Smith (http://home.mira.net/~galap) wrote some code (Java and Perl) to manipulate files such as these. They were originally intended just as simple configuration files (astute readers will have noticed this use in the above code) but have since been perverted to a quick and dirty database. Thanks Craig!

This is an example of what you might find on a running system in '/localhost/opt/kara/var/staging/localhost/config'.

```
etc/ntp.conf=list {
  U
  todo-pre
  1084430056
}
```

The first line obviously represents the filename in question. The first element of the array represents the notification from CVS ("U" for "Updated"). The second is the status of the update and the third is an epoch-second timestamp for the last action taken.

6.4 Update state transitions

The process of installing the update involves three distinct major states and three closely-related error states:

'todo-pre'
: Initial state, ready to run 'pre.sh'. As set by 'cvsup-buildprops'.

'todo-pre-error'
: An error occurred while running 'pre.sh'.

'todo-install'
: The installation script for this file should be run.

'todo-install-error'
: An error occurred while running 'install.sh'.

'todo-post'
: The post-installation script for this file should be run.

'todo-post-error'
: An error occurred while running 'post.sh'.

Note that there is no automated process for recovering from an error condition (although you could of course implement one). The general procedure would be to resolve the error condition and reset the relevant state to 'todo-pre'.

6.5 Example "uber-script" - part 2

This script does the real work of running the action scripts. It has some notable bugs, not the least of which is the fact that it doesn't handle removed files. However, it's survived several years of production use largely intact, so it can't be all bad. 

This is currently in the repository as 'kara/release/3.5/opt/kara-tools/update-config'.

```
#!/usr/bin/perl

use strict;
use FileHandle;
use CS::Properties();

# Get config
my $configprops = new CS::Properties("kara-config.props");
my $CONFIGFILECHECKOUT=$configprops->get("CONFIGFILECHECKOUT");
my $CONFIGACTIONCHECKOUT=$configprops->get("CONFIGACTIONCHECKOUT");
my $cvsworkfile=$configprops->get("cvsworkfile");
my $default_installscript=$configprops->get("default_installscript");

# read existing CVS output
my $cvsoutputprops = new CS::Properties($cvsworkfile);
$cvsoutputprops->read($cvsworkfile);

# Process list
my $cvs_file;
foreach $cvs_file ($cvsoutputprops->keys()) {
```

```
    my @item = $cvsoutputprops->get($cvs_file);
    my $cvs_action = $item[0][0];
    my $cvs_file_status = $item[0][1];
#   print STDERR "$cvs_action $cvs_file $cvs_file_status\n";


  CASE: {

    ($cvs_file_status eq "done") && do {
      # no update required
      last CASE;
    };

  # Things OK - process this puppy!
    # File was updated
    ($cvs_action eq "U") && do {
      print "* Updated - $cvs_file\n";
      doPreAction($cvs_file,$cvs_action,$cvs_file_status);
      doInstall($cvs_file,$cvs_action,$cvs_file_status);
      doPostAction($cvs_file,$cvs_action,$cvs_file_status);
      last CASE;
    };

    # File updated, local copy patched instead of downloading new file
    ($cvs_action eq "P") && do {
      print "* Patched - $cvs_file\n";
      doPreAction($cvs_file,$cvs_action,$cvs_file_status);
      doInstall($cvs_file,$cvs_action,$cvs_file_status);
      doPostAction($cvs_file,$cvs_action,$cvs_file_status);
      last CASE;
    };

    # Diffs between local copy and repository version merged OK
    ($cvs_action eq "M") && do {
      print "* Merged - $cvs_file\n";
      doPreAction($cvs_file,$cvs_action,$cvs_file_status);
      doInstall($cvs_file,$cvs_action,$cvs_file_status);
      doPostAction($cvs_file,$cvs_action,$cvs_file_status);
      last CASE;
    };

    # Additions/removals
    # New file.  New service?
    ($cvs_action eq "A") && do {
      print "* Added - $cvs_file\n";
      doPreActions($cvs_file, $cvs_action, $cvs_file_status);
      doInstall($cvs_file,$cvs_action,$cvs_file_status);
      doPostActions($cvs_file,$cvs_action,$cvs_file_status);
      last CASE;
    };


    # Ahem... this is just wishful thinking as CVS never says "R".
    ($cvs_action eq "R") && do {
      print "* Remove - $cvs_file\n";
      doRemove($cvs_file,$cvs_action,$cvs_file_status);
      last CASE;
    };

  # Bad things happened - stop and tell a human!
    # Merge attempted but failed - conflicts must be resolved by hand
    ($cvs_action eq "C") && do {
      print "* Conflict - $cvs_file\n";
      print "Aborting due to conflict!\n";
      exit 1;
    };

    ($cvs_action eq "?") && do {
      print "* Unknown - $cvs_file\n";
      print "Aborting due to unknown file!\n";
      exit 1;
    };
```

```
    # Blank or otherwise unrecognised line - shouldn't happen
    print "* Aborting - unable to parse: $_\n";
    exit 1;

  } # end of switch

} # end of main program

############################
# Action processing

sub doPreAction {
  my ($cvs_file, $cvs_action, $cvs_file_status) = @_;

  my $actionscript = "$CONFIGACTIONCHECKOUT/$cvs_file/pre.sh";
  my @actionargs = ("$CONFIGFILECHECKOUT/$cvs_file", "/$cvs_file");

  # check to see if pre.sh script exists.  If not, not fatal.
  if (-x $actionscript) {
    my $result = system($actionscript, $actionargs[0], $actionargs[1]);
    if ($result != 0) {
        changeStatus($cvs_file, $cvs_action, "todo-pre-error");
        die "Error in:\n $actionscript (@actionargs)\n";
    }
    else {
        changeStatus($cvs_file, $cvs_action, "todo-install");
    }
  }
  else {
    print STDERR "$actionscript not found\n";
    changeStatus($cvs_file, $cvs_action, "todo-install");
  }
}

sub doInstall {
  my ($cvs_file, $cvs_action, $cvs_file_status) = @_;

  # Check for install.sh.  Use default if it doesn't exist.
  my $actionscript = "$CONFIGACTIONCHECKOUT/$cvs_file/install.sh";
  my @actionargs = ("$CONFIGFILECHECKOUT/$cvs_file", "/$cvs_file");

  if (-x $actionscript) {
    my $result = system($actionscript, $actionargs[0], $actionargs[1]);
    if ( $result != 0 ) {
        changeStatus($cvs_file, $cvs_action, "todo-install-error");
        die "Error in:\n $actionscript (@actionargs)\n";
    }
  }
  elsif ( -x $default_installscript ) {
    my $result = system($default_installscript, $actionargs[0], $actionargs[1]);
    if ($result != 0 ) {
        changeStatus($cvs_file, $cvs_action, "todo-install-error");
        die "Error in:\n $default_installscript (@actionargs)\n";
    }
  }
  else {
    print "Can't find $actionscript or $default_installscript \
        - quitting!\n";
    changeStatus($cvs_file, $cvs_action, "todo-install-error");
    exit 1;
  }
  # Mark install done.
  changeStatus($cvs_file, $cvs_action, "todo-post");
}

sub doPostAction {
  my ($cvs_file, $cvs_action, $cvs_file_status) = @_;

  # check to see if post.sh script exists.  If not, not fatal.
  my $actionscript = "$CONFIGACTIONCHECKOUT/$cvs_file/post.sh";
  my @actionargs = ("$CONFIGFILECHECKOUT/$cvs_file", "/$cvs_file");
  # check to see if post.sh script exists.  If not, not fatal.
  if (-x $actionscript) {
```

```
    my $result = system($actionscript, $actionargs[0], $actionargs[1]);
    if ($result != 0) {
        changeStatus($cvs_file, $cvs_action, "todo-post-error");
        die "Error in:\n $actionscript (@actionargs)\n";
    }
  }
  else {
    print STDERR "$actionscript not found\n";
  }
  # Mark post-action done.
  changeStatus($cvs_file, $cvs_action, "done");
}

sub doRemove {
  my ($cvs_file, $cvs_action, $cvs_file_status) = @_;

  # Check for remove.sh.  Fail if it doesn't exist!
  my $actionscript = "$CONFIGACTIONCHECKOUT/$cvs_file/remove.sh";
  my @actionargs = ("$CONFIGFILECHECKOUT/$cvs_file", "/$cvs_file");
  if (-x $actionscript) {
    if (system($actionscript, $actionargs[0], $actionargs[1]) != 0 ) {
        changeStatus($cvs_file, $cvs_action, "todo-remove-error");
        die "Error in:\n $actionscript (@actionargs)\n";
    }
  } else {
    print STDERR "$actionscript not found\n";
    changeStatus($cvs_file, $cvs_action, "todo-remove-error");
    exit 1;
  }
  # Mark remove-action done.
  changeStatus($cvs_file, $cvs_action, "done");
}

####################
# Helper subroutines

sub changeStatus {
  my ($cvs_file, $cvs_action, $cvs_file_status) = @_;
  $cvsoutputprops->put("$cvs_file", [ $cvs_action, $cvs_file_status, time() ]);
  $cvsoutputprops->write($cvsworkfile);
}
```

7. Scheduling Kara

You'll probably want to automate the running of Kara, which you might do via 'cron'. Note that besides running the two uber-scripts, you might also want to automatically update your Kara action scripts from the master repository.

```
### Kara updates
#
# update local config
*/15 * * * * cd /localhost/opt/kara/libexec/\
release/3.5/opt/kara-tools && \
./cvsup-buildprops && ./update-config
# update Kara tools
0 1 * * * cd /localhost/opt/kara/libexec/\
release/3.5 && cvs -q up -d
```

8. Kara and File Permissions

Kara is unfortunately pretty dumb when it comes to file ownership and permissions. Some of this is due to the CVS filter through which each file passes. The default install script will happily install files as 'root', using the default umask, so you'll probably wind up with files of mode 644.

In the case where files contain sensitive information you don't want every user on the system to see, you should create a specific action script for the file that sets appropriate permissions (create a dummy, set permissions, _then_ copy the file). Indeed, some software subsystems perform checks on their configuration files and won't run if they're "wrong", so this can be quite important, even for an essentially single-user box like a firewall.

Likewise, since Kara mostly runs as root in order to do its work, slack controls on your CVS repository can lead to a security nightmare. Lock it down!

9. Kara and Backups

There's a great line in the BSDI manual - something like "If you don't do backups, you'll be sorry." This is as true today as it was then.

Backing up the master action script repository should be pretty obvious - just take a copy of '/localhost/opt/kara/svc/cvs' (or whereever you put it). You'll have copies of the files themselves in checkouts on each node, but if you lose the repository you'll also lose all your file modification history.

Likewise, you'll probably have checked-out copies of configuration files on systems other than the nodes themselves, but you should also arrange to back up each node's configuration repository. It might even make sense for you to "rsync" them back to a master machine and backup from there. Or perhaps even do that from a couple of "masters"…

10. Klusters?!

While Kara is largely designed with notion of managing of one system per configuration repository (so that each individual system can be self-reliant), there is provision for managing multiple nodes from a master node.

You may have noticed that we place our staging checkout in '/localhost/opt/kara/var/staging/localhost/config'. It should be a relatively simple matter to substitute hostnames for the second 'localhost' in that path and modify the uber-scripts to iterate over the multiple directories (or just run multiple copies with different configuration files).

11. Kara and system patches

In the initial design phase, we concluded that configuration management and system patch management were completely separate problems and decided to concentrate on the former. However, as it turns out, Kara lends itself to system patching as well.

This can work simply by having a separate CVS tree (per operating system release), containing an overlay of any patches you want to apply to the system. This could go in a master CVS respository, since it is probably not as critical to have available full-time as the configuration repository. Check out your patch tree on each system and the standard Kara tools can be used to perform the updates.

12. Conclusion

We hope that this system is of use to you (all the code, such as it is, is available under a BSD licence). At the very least, we hope it sparks some thought about configuration management. We'd be very happy to hear feedback, suggestions for improvement, offers of assistance or even just a quick note to let us know you're using Kara (or some form thereof).

In conclusion, probably the best advice is to remember that Kara is just a tool. For it to be effective, *you* must adopt a certain discipline. If you don't use Kara, Kara can't help you (and maybe it's fine in some cases just to do ad-hoc edits of configuration files). If you do use it, the cost is time, but the benefit is (hopefully) sanity.

References

You may find the latest version of this document at http://www.fernhilltec.com.au/~adrian/kara/, along with more specific installation instructions.

Acknowledgements

- Michael Paddon
- Craig Smith
- Neal Wise
- Malcolm Herbert
- The OpenBSD team
- The CVS guys
- Andrew Tridgell (for 'rsync')
- The SSH guys
- Staff guinea pigs and customers of Fernhill Technology.