The following paper was originally presented at the
Ninth System Administration Conference (LISA '95)
Monterey, California, September 18-22, 1995

Patch Control Mechanism for Large Scale Software

Atsushi Futakata
Central Research Institute of Electric Power Industry (CRIEPI)

Patch Control Mechanism for Large Scale Software

Atsushi Futakata – Central Research Institute of Electric Power Industry (CRIEPI)

ABSTRACT

Applying patches to large scale software is often difficult because unofficial patches and user modifications conflict with any "official" patches. Version control systems such as RCS[1], CVS[2], and configuration management[3,4,5] are useful solutions for this problem when the baseline of the software is fixed. However, an official patch that is developed externally changes the baseline and any local changes based on this become obsolete. Thus we must re-apply various unofficial patches and modifications, identify the causes of conflict, change or remove patches, and repeat the patch and unpatch operations.

This paper presents a mechanism for (1) managing versions of a software package based on patches, (2) automating the application of unofficial patches and modifications by the user, and (3) rebuilding the package using file versions instead of timestamps. Using this mechanism, it becomes easy to apply patches and re-build software.

Introduction

We have spent a lot of time installing and patching large scale software packages such as the X11 Window System, TeX, etc. Installation of new software involves checking storage space, reading documentation and setting various configuration files correctly. This can be a non-trivial task even if the platform is officially supported. If the platform is not supported, installation becomes more complicated because changes to the source code may be required and tools such as *Configure* are not applicable. Thus software porting systems represented by the FreeBSD ports system[6] appear and become to support the installation task.

Applying patches poses another difficult problem: If only official patches are applied to an officially supported platform, the task is usually easy because the patches are well managed and cause no conflict. However an unsupported platform requires source code changes which often conflict with an official patch. Furthermore, the user may require many useful, unofficial patches. These may be patches for emergency security, localization (e.g., japanization), machine/OS-dependencies or various extensions, such as Tcl/Tk has. Those patches may also conflict with official ones. The reason for the conflict is a lack of version management facilities for distributed development. This conflict usually necessitates the following operation:

- Remove all unofficial patches and apply the official one,
- Re-apply the unofficial patches and user modifications. If reject files are generated, the unofficial patch must be fixed or removed,
- Rebuild the software. This can take a long time because the above operations may cause unnecessary changes to timestamps.

Configuration management systems such as Aegis[7], CMS/MMS[8] are useful for version control and building software for multi-user development. They target the continuous development of the software and manage products based on a current baseline, that is a reference version of software on which each member of developing team fixes bugs and develops new functions. After each task is complete, all modifications are integrated and the modified source code becomes a new baseline for succeeding development. This baseline approach is useful for inhouse development teams.

However, an official patch is delivered outside of a user's control and it only changes the baseline. All modifications based on the previous version of the software then become obsolete. Thus if the user wants to apply a new official patch, all other patches and modifications must be rearranged and re-applied after the official one is applied.

In future, self-adaptive software agents or automatic programming from very high level specifications may solve the problem but, for the present, we have no silver bullet. Thus, in order to solve the above problem and support patch application, this paper proposes a patch control mechanism which has the following features:

- version management of the whole package, including individual files and patches,
- management and control of patch application order,
- assistance with patch/unpatch operations and patch modification,
- software rebuilding according to an individual file version rather than a timestamp.

Classification of patches

Unofficial patches may be generated by different people based on different baselines. Unified management of these patches can be difficult and confusing. In this paper, we classify patches into the following three types and treat each differently.

official patch
: A patch that is authorized by and distributed from the software developer/maintainer. The latest official patch number is the official software version and we call it the patchlevel.

unofficial patch
: A patch/extension that is widely distributed but is not an official patch. An unofficial patch may be applied in various directories with various *patch*(1) options.

modification
: A change made by the local user, which includes editing files, fixing bugs, changing configuration files, etc.

System Overview

This section presents an overview of the system which is an implementation of the patch control mechanism. This includes; (1) management of the three types of patch, (2) control of patch application, and (3) rebuilding of the software. The components *VM (Version Manager), PM (Patch Manager),* and *BM (Build Manager)* implement the three functions respectively. Figure 1 shows the components and the relation among them.

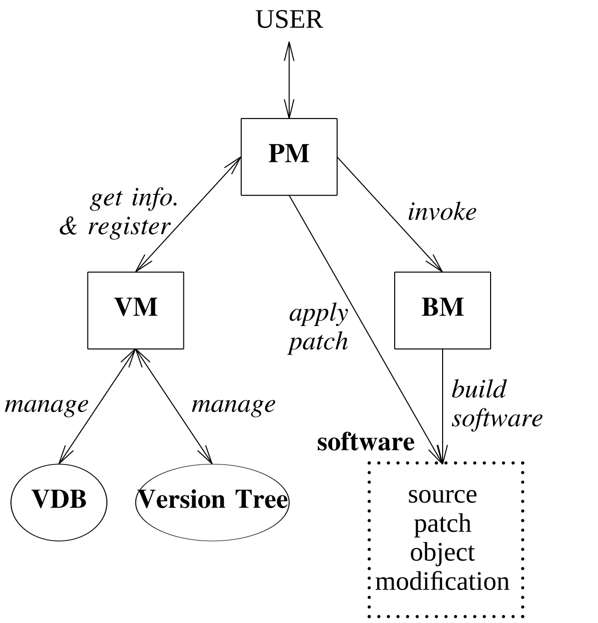

Figure 1: The components of this system

The VM manages information in the *VDB (Version Database)* and the *version tree*, which records information about version control for updating and rebuilding of the software. The VDB records the location of the patches and the versions of individual files. The version tree records the application order for unofficial patches and modifications at each patchlevel. When a new patch arrives, the user adds the patch to the VDB using the VM. If the user wants to apply this, and the result is successful, the VM registers the sequence of patches actually applied, to the version tree.

The PM (Patch Manager) controls the patch/unpatch operations and the building of the software according to the version tree. In this system, all operations, including editing a patch file, applying a patch, and building a package, are performed via the PM, and the result of the operation is reflected in the VDB and the version tree.

When a user applies a new official patch, the PM tries to apply unofficial patches which were applied to the last version of software. If one of the patches is rejected, the PM notifies the user. The user may then remove or edit this patch and continue the job. After the job finished, the PM returns the result of the patch application and the VM revises the VDB and the version tree.

The BM (Building Manager) is an extended *make* command, invoked from the PM to build a target according to the version of file instead of its timestamp. Because a new official patch forces patch/unpatch operations of unofficial patches and modifications, the timestamp of a file may change even if the contents is not altered. The VM registers the version of the newly generated target with the VDB and the version tree.

This system manages several packages at once by referring to the **pcm** file. An entry of the **pcm** file has the following form:

application_name:top_directory:patch_option

Application_name is an identifier to be used for selecting a package. *Top_directory* is the directory where official patches are applied. *Patch_option* species an option to the *patch*(1) command. For example, the entry for X11R5 (X11 Window System, Version 11, Release 5) becomes[1]:

```
X11R5:/X11R5:-p -s
```

This means that the following command is needed to apply the patch.

```
% cd /X11R5
% patch -p -s <foo.patch
```

Version Management

In this system, changes of source codes and the source code itself are managed separately to make

[1]Because of the limitation of line width, we denote the location of the X11R5 package as `/X11R5` in the following examples. The actually location is `/staff/src/X11R5` in our site.

patches independent of the baseline specified by an official patch. Thus each unofficial patch and modification has a separate version to avoid conflicts which may occur with any new official patch. The versions of these patches are managed using RCS.

The VM manages the patch information, including the location of patches, versions of patches themselves and the history of patch application. The VM performs the following functions:

- generates the specified version of a software package or a file by applying patches automatically,
- registers/deletes/updates a patch,
- creates/updates a modification from the differences between a modified file and its original,
- maintains versions of files, which are determined by the patches which actually cause change.

Information managed by the VM is recorded in the VDB and the version tree. The VDB consists of the locations and the versions of patches and the versions of the individual files. The version tree describes the order of patch application required to make the specified version of software, and which version of each patch should be applied.

The version of the software package itself is represented by a path of the version tree. For example, `#3:@1.2,@2.1,@3.2:$1.2` means that the version is generated by application of an official patch `#3`, unofficial patches `@1.2`, `@2.1`, and `@3.2`, and a modification `$1.2` in order. The following section describes the contents of the VDB and the version tree.

```
#26
#1:/X11R5/fixes/fix-01
#2:/X11R5/fixes/fix-02
#3:/X11R5/fixes/fix-03
#4:/X11R5/fixes/fix-04
```

Figure 2: A part of .official file for X11R5

Version Database

The VDB consists of the four files.

.official

`.official` contains the current patchlevel and the locations of official patches. The first line is the current patchlevel of the software. The following lines contain a patch identifier (which is used in the version tree), and a corresponding patch location. Figure 2 is a sample of a part of a `.official` file. `#26` in line 1 means that the current patchlevel is 26. The lines 2-5 specify the location of each official patch. For example, line 2 means that the location of the official patch `#1` is `/X11R5/fixes/fix-01`.

.unofficial

`.unofficial` contains the locations of unofficial patches and the information required to apply them. Each entry of this file has the following form;

id:location:place:option

Id is the unofficial patch identifier which is used in the version tree. *Location* is the location of the unofficial patch. *Place* is the directory in which the patch is applied, and *option* is the *patch*(1) options. In general, there is no standard method for applying unofficial patches, and this is a reason for the *place* and *option* fields. Figure 3 shows a part of `.unofficial` for X11R5.

```
@1:/X11R5/fixes/Xaw-p1:/X11R5:-p0
@2:/X11R5/fixes/Xsi-p1:/X11R5:-p0
@3:/X11R5/fixes/Xwchar-p1:/X11R5:-p0
@4:/X11R5/fixes/Xaw-p2:/X11R5:-p0
```

Figure 3: A sample of .unofficial file for X11R5

Versions of the patches are managed by RCS and the RCS file for each patch is located in *directory of location/RCS*. A user can edit the patched files themselves instead of the patch because it is almost impossible to edit the patch directly. Changes to the files are reflected in the patch by the following process:

- choose the version of the software and the target patch to be edited. For example, we assume that the patch is `@3.1` which changes two files, `foo.c` and `bar.c`, and the version is `#3:@1.2,@2.1,@3.1`,
- apply the sequence of patches which should be applied in this version before applying the target patch. After that, make a copy of the patched file and apply the target patch. In this example, first, the VM applies `@1.2` and `@2.1` to the software whose patchlevel is `#3`. Next, the VM makes copies of `foo.c` and `bar.c` with an extension *.prev*. Then, the VM applies `@3.1` to the software,
- after editing the patched files, make a new patch by running *diff*(1) against the files of which the VM made copies in the last step. In this example, the two diff files between `foo.c/bar.c` and `foo.c.prev/bar.c.prev` are concatenated to a new patch, whose path name is the same as `@3.1`.
- check in the new patch using *ci*(1) . In this example, the VM runs the following command:

  `% ci -r2.1` *location of @3*

  in which 2 of `2.1` is the new version number for the patch `@3`.

The VM normally uses only the release number of RCS. Thus a patch with version *N* has a revision *N.1* in the RCS file. For example, when applying the version 3 of the patch `@1` in the Figure 5, the following commands are needed:

```
% co -l -r3.1 /X11R5/fixes/Xaw-p1
% cd /X11R5
% patch -p0 </X11R5/fixes/Xaw-p1
```

.modification

`.modification` contains the locations of user modifications. There is at most one modification file per directory and the result of editing the source is reflected in the modification file in the same way as unofficial patch files. Figure 4 shows a part of `.modification`. The first field is the identifier of modification and the second specifies the location of the modification.

```
$1:/X11R5/mit/config/config.patch
$2:/X11R5/mit/lib/Xt/Xt.patch
```

Figure 4: A sample of .modification file for X11R5

.f_ver

`.f_ver` contains the version history of the source and object files. In this system, a file has two different forms of version. One is the strict version which is indicated by the software version. The other is the historical version which indicates the history of changes by patches. The historical version is used instead of the file timestamp when the software is rebuilt. For example, the strict version of file `foo` is indicated as:

```
foo.#3.{@1.1,@2.1,@3.2}.$1.2
```

where `#3` means the official patchlevel is 3 and *@N.M* means that the applied unofficial patch identifier is *N* and the version of the patch itself is *M*. `$1.2` means that the version of a modification to `foo`. The historical version has the following form:

```
foo:#1,#3:@1.1,@2.1:$1.2
```

This means that `foo` is changed by the official patches `#1` and `#3`, unofficial patches `@1.1` and `@2.1`, and user modifications with version `$1.2`.

`.f_ver` must exist in all subdirectories of the software source tree. An source/object entry in this file is updated as follows:

- if a patch is applied to the file, the identifier of the patch is added to the entry,
- if the version of a source file differs from the object file, after making the object, the object is given the same version as the source. If multiple sources exist, e.g., linking *.o files, the versions of the sources are merged and becomes the version of the object because it is made under the effect of patch applications to the sources. This method is described in the section **Make Command**,
- editing the file changes the version of the modification in the entry.

If the file in the VDB is a symbolic link, the VM follows the link and updates the location of the file to be the real location.

Version tree

The version tree manages the software version and describes the application order of unofficial patches and modifications at each patchlevel. The version tree includes applied unofficial patches and modifications only. Figure 5 shows the concept of the version tree. *#N* is the patchlevel and *@N.M* and *$N.M* are the unofficial patch identifier and the modification identifier to be applied. A conflict between unofficial patches causes branching or modification of a patch.

The `.vtree`, which is located in the top directory of the software, records the version tree as the collection of the following form;

official_id:unofficial_ids:modifications

For example, the path **A** in figure 5 is described as "**#1:@1.1,@2.1,@4.1:$1.1,...**" and the path branched from `@1.1` is described as "**#1:@1.1,@3.1, @4.2,...**".

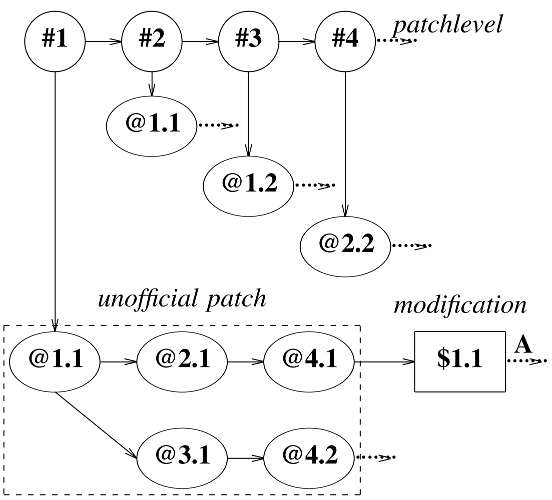

Figure 5: The concept of version tree

Patch Control Mechanism

This section describes the patch control mechanism based on the version tree. The PM stores source files to which only official patches are applied. The unofficial patches/modifications are applied on demand when editing the latest version sources, rebuilding the software, etc.

The PM provides an asynchronous way to apply a patch or to rebuild a software package by exchanging information with the user via e-mail. Once the PM is invoked, the PM reports conflicts or compilation failure to the user via e-mail. After the user edits files or abandons the patch, the user only sends a simple command with the file contents if it is needed. The PM accept the following commands:

- **edit** [*file* | *id*] *(ver)*
  edit the *file* or a patch whose identifier is *id* in the software version *ver*. In the PM interface,

a user can specify the *ver* in the form given in the version tree, or as the entry number in the version tree which the user can obtain by sending the command **scan -v**. This causes a change to an unofficial patch or a modification.

- **add [-o |-u]** *patch_info*
  register an official/unofficial patch with the *patch_info*. The *patch_info* has the same form as in `.official` or `.unofficial` without the identifier. The identifier is added by the VM.
- **scan [-v |-o |-u |-m]**
  show the contents of `.vtree`, `.official`, `.unofficial`, or `.modification`.
- **show** *id (ver)*
  show the contents of patch whose identifier is *id* in the software version *ver*. If the *ver* is omitted, **show** shows the patch that has been applied in the current version of software, or the latest version of the patch if the current version of software does not include this.
- **apply** *id* (*ver*)
  apply a patch whose identifier is *id*. If the patch is an official patch, a user does not need to specify the *ver*. If the patch is an unofficial patch or a modification, the user must specifies the *ver* as the full path form. Processing of the **apply** function is described later.
- **make [-T]** (*ver*)
  rebuild a software package whose version is *ver*, or the current version if the *ver* is omitted. If a user wants to rebuild the package according to timestamps, **-T** option is necessary.
- **grep [-o |-u |-m |-a]** *string*
  grep *string* from official patches, unofficial patches, modifications and all patches.

When **apply** or **make** is issued, the PM interacts with the user. When applying a new official patch, the PM performs the following actions:

1. apply the official patch.
2. make a sequence of unofficial patches to be applied. The PM extracts the last sequence from the version tree and notifies the user via e-mail. The user can remove several patches or change the application order. The user then replies by e-mail.
3. apply unofficial patches and any modifications in order. If no reject files are generated by *patch*(1), step (3) is done. Otherwise the PM suspends the job and sends the patched file with the rejected part embedded at a suitable place to the user via e-mail. The user must decide either to edit and apply this patch or to abandon this. If the user decides to abandon, the PM strips this patch from other files to which the patch was applied and the PM applies a next patch. The user can edit it before replying to the PM. In this operation, the user can refer to the whole patch by sending a command to the PM. When the all replies are gathered, the PM replaces the parts of the patch and reflects the change in the RCS file using *ci*(1). After that, the VM tries to apply the next patch.
4. create a new entry in the version tree. If user wants to rebuild the software, the PM invokes the BM.
5. unpatch all the unofficial patches and the modifications.

When applying a new unofficial patch, the user should specify the version tree entry which includes the new patch. This new version tree entry is registered if the application process has terminated successfully. The other operations are similar to the above case for official patches.

If reject files are generated, the PM sends e-mail to the user per reject file. Figure 6 is a sample of the e-mail, which notifies that application of unofficial patch `$1.1` has been failed in `sun.cf`[2]. The body of e-mail is the contents of `sun.cf` and the rejected file is embedded in it.

```
To: futakata
Subject: REJECTION $1.1: sun.cf
From: PCM <pcm@denken.or.jp>

<Contents of sun.cf continue.>
    ......
```

Figure 6: A head of e-mail which notifies the rejection

After receiving the e-mail, the user can choose from the following three actions:

- **replace**: replaces a part of patch/modification by this file,
- **delete**: deletes this patch from the version tree entry,
- **abort**: aborts the job, recovers the software and terminates the PM.

If the user chooses **replace**, the user must reply the e-mail like Figure 7 to the PM after editing the file in the original e-mail. The first line of the mail body is the action that the user chosen, and the rest of the body is the file to be replaced. If the user chooses **delete** or **abort**, only the action is needed in the body. After receiving all replies from the user, the PM extracts a new patch from the replies by the same manner described in the explanation of `.unofficial`, and applies the new patch instead of the original patch and revises the version of the patch.

[2]We abridge the file name in the subject field of Figure 6 for convenience of display. The subject of real e-mail contains the full path name of the file.

```
To: pcm@denken.or.jp
Subject: RE: REJECTION $1.1: sun.cf
From: futakata@denken.or.jp

replace
<Contents of new sun.cf continue.>
    ......
```

Figure 7: A head of reply mail with the action **replace**

Make Command

The PM often causes needless changes to timestamps and confuses timestamp-based traditional *make*(1) commands. Thus we developed the BM, which is based on the *GNU make* command, to perform actions according to the historical file version, i.e., when the historical versions of source files are not included in the version of the object, the BM rebuilds the object. After that, the BM revises the version of object by the merged version of sources because the object is made under the effect of all patches applied to the sources.

To compare versions, the BM treats the version as an ordered sequence of patches. Thus a version *A* includes a version *B* if and only if all patches in the *B* is appeared in the *A*, and an unified version of *A* and *B* does not causes inconsistency of the application order. For example, the version `#1:$1.1,$2.1,$3.1` does not include the `#1:$3.1,$1.1`.

After building objects, the BM sends the result to the VM and the VM updates the historical versions of the objects.

Experiment

We have tested this system for applying patches to the X11R5. In this experiment, the number of official patches is 26 and the number of unofficial ones is 12 including internationalization patches for libraries Xaw, Xsi and Xwchar. We also modify configuration files in the `/X11R5/mit/config` directory and several files in `/X11R5/mit/lib/Xt`. Figure 8 is a head part of the version tree of this experiment.

After the official patch `#3` is applied, 3 unofficial patches are issued and we apply them and rebuild the X11R5. There is no problem in this phase. When the `#6` is applied, the internationalization mechanism of X11R5 is changed and `@1.1` and `@2.1` become obsolete. The PM then sends e-mail telling us that reject files are generated. We remove the patches and do not rebuild X11R5. After `#8`, a new unofficial patch `@4.1` is issued and we rebuild after applying `@4.1`.

This experiment shows that it is easy to control the patch/unpatch operations and to find conflicts between official patches and unofficial ones.

```
#0::$1.1,$2.1,$3.1,$4.1
#1::$1.1,$2.1,$3.1.$4.1
#2::$1.1,$2.1,$3.2.$4.1
#3:@1.1.@2.1,@3.1:$1.2,$2.1,$3.3,$4.1
#4:@1.1,@2.1,@3.1:$1.2,$2.1,$3.3,$4.1
#5:@1.1,@2.1,@3.1:$1.2,$2.1,$3.3,$4.1
#6:@3.1:$1.2,$2.1,$3.3,$4.1
#7:@3.1:$1.2,$2.1,$3.3,$4.1
#8:@3.1,@4.1:$1.3,$2.1,$3.3,$4.1
```

Figure 8: A part of the version tree for X11R5

Conclusion

In this paper, we presented a control mechanism for applying patches and confirmed that this system works well. However, even if the patches are successfully applied, rebuilding the software still takes a long time. In order to reduce the rebuilding time, we have tried to develop a mechanism for analyzing the dependency between makefiles and a distributed make mechanism based on it.

Availability

The system which we presents in this paper will be available via WWW from http://www.denken.or.jp/people/cirl/futakata.

Acknowledgements

I would like to thank Paul Anderson at the Univ. of Edinburgh for his useful advice on drafts of this paper. My thanks also go to Yasusi Sinohara and Shouichi Matsui at the CRIEPI for their suggestion in development of this system.

Author Information

Atsushi Futakata graduated from the Tokyo Institute of Technology at Tokyo in 1987. He has made studies of automatic programming and management of distributed information system in the Central Research Institute of Electric Power Industry. He is now working on the CSCW environment for researchers.

Reach him via mail at: Information Science Department, Communication and Information Research Laboratory, CRIEPI, 11-1 Iwado-kita 2-chome Komae-shi Tokyo 201, JAPAN

His e-mail address is futakata@denken.or.jp.

References

[1] Tichy, W. F.: "RCS - A System for Version Control", *Software-Practice & Experience*, Vol. 15, No. 7, 1985.

[2] Berliner, B.: "CVS II: Parallelizing Software Development", included in the *CVS ver.1.3* package, 1995.

[3] Bersoff, E.: "Elements of Software Configuration Management", *IEEE Transaction*

of Software Engineering, Vol. SE-10, No. 1, 1984.

[4] Whitgift, D.: *Methods and Tools for Software Configuration Management*, John Wiley and Sons, 1991.

[5] Eaton, D.: "comp.software.config-mgmt FAQ version 1.12", posted to the `comp.software.config-mgmt`, 1995.

[6] Palmer, G. and Hubbard, J. K.: "The FreeBSD Ports FAQ file version 1.1", included in the *FreeBSD rel.2.0.5* package, 1995.

[7] Miller, P.: "Aegis: A project Change Supervisor User Guide", included in the *aegis ver.2.1* package, 1993.

[8] DEC: *CASE Environment of DEC: COHESION - COHESION handbook rev.1*, DEC, 1991.