Beam: A Tool for Flexible Software Update

Thomas Eirich – University of Erlangen-Nürnberg, Germany

ABSTRACT

Today's workstations often have a limited local disk space. Besides putting the home of the workstation's owner onto the local disk it is reasonable to place frequently used software packages on the disk, too. This reduces network traffic and makes a workstation more independent from file servers. Of course, the replicated software must be kept consistent with the versions on the file servers. This should be done by an automatic update mechanism.

Copying software packages in their entirety would quickly fill up the local disk space. Especially this problem is addressed by *Beam*. Copying the whole software package is merely the simplest form of *Beam*'s update possibilities. A system administrator can rely on powerful features for writing update scripts: merging of several source trees, enhanced file name generation, embedded Perl code, a rich set of update commands which can be arbitrarily combined to form complicated update rules. Additionally, *Beam* has a PACK concept which allows easy adaptation of the update process to the usage pattern of a workstation's owner. To save space on the local disk the user can omit those parts of software packages which are not needed at all (e.g., foreign language user interface) or which are of less interest (e.g., manuals for experienced users). These parts are not missing on the workstation because a symbolic link to the server version is inserted.

Introduction

The original initiative to develop *Beam* arose from a typical situation in workstation clusters. Workstations often have disks of limited size (300-500MB). Besides putting the homes of the main users on the local disk it would be reasonable to add frequently used software packages. This has the advantage of reducing network load and making workstations more independent from file servers. If a server is down it is more likely that the workstation will survive the downtime without hanging when most of the daily used software is locally available. Furthermore, it is possible to disconnect a workstation from the net and operate it stand-alone because most of the tools with the majority of their functionality have been placed onto the local disk.

Simply copying software packages from servers to workstations is not a good idea. First, the required space would quickly exceed the limits of the local disk. Second, changes made on the server after the copy are not propagated to the clients. Soon, each workstation has different software versions and errors occur because programs do not fit together anymore.

Thus, the idea of *Beam* was born. *Beam* copies software packages and keeps them consistent with the source version. At first glance, this could also be achieved with existing tools like rdist [1], but *Beam*'s strength arises from the flexibility and easy customization performing these updates, e.g., several sources can be merged to a single destination.

offers several update commands which modify files while they are updated. There are commands to change ownership of files, to modify the contents of files, to transform symbolic links, and many others. Complicated and powerful update rules can be constructed by composing update commands. Furthermore, most update commands can be customized by Perl [6] code embedded in *Beam* instruction files.

Groups of files and their update rules can be associated with a symbolic name. Such a group is called a PACK. By composing a list of PACK names, the user can easily customize the update process. Customization often requires to exclude parts of some software from update in order to save disk space. These parts are still accessible because symbolic links to the server version are inserted.

In the following chapter we outline the update concept of *Beam*. Then, we present details of *Beam* instruction files and some of the most important update commands. The updating of a cluster of workstations then shows one possible application of *Beam*. Finally, current and related work is discussed.

The Update Concept of Beam

Software packages are viewed as file trees. The update process of software packages is controlled by an instruction file. This file defines the source path of the software package, the destination path, and associates update rules with path names. The path names are specified relative to the source or destination path. An update rule consists of update commands. Arguments can be supplied to

```
$x11 = "X11R5"              if $x11 eq "";
$src = "/local.remote/$x11" if $src eq "";

sub dst {
     foreach $d ("/usr/local.stand/$x11", "/local.stand/$x11") {
                return $d if -d $d; } }

sub mainMajor {         # Select for each shared library and each major
        ... }           # version the one with the highest minor version

sub DPI { ... }         # Find out the dpi of the monitor (75/100)

FROM:       $src
TO:         &dst
NOTIFY:     mail    warning  @users
            file    update   localhost   "/usr/tmp/beam:$src->&dst"
            syslog  error


PACK:   std                            # standard version
          **/*.{old,orig}             : delete
          lib/                        : net
             lib*.{so,sa}.*           : fgen filter=mainMajor : syn
             X11/
                X*DB rgb.*            : syn
                fonts/                : net
                   &{DPI}dpi          : syn
                   misc/              : syn
                           hangl*     : net    # Very big font files
                           jiskan*    : net    #   but never used
                   /
                ...
          /  /  /
          ...

PACK:  -misc-fonts                    # no misc fonts at all
          lib/X11/fonts/misc          : net

PACK:  +misc-fonts                    # all misc fonts
          lib/X11/fonts/misc          : syn

PACK:  +cc-kit
          include/X11                 : syn
          bin/{imake,makedepend,xmkmf} : syn
          lib/*.a                     : syn

PACK:  light-cc                       # smallest stand-alone vers w/ C stuff
          <std>                       : include
          <-misc-fonts>               : include
          <+cc-kit>                   : include

PACK:  subserver  # for a certain architecture (argument: arch=XXX)
           ...
```

Figure 1: Sketch of an instruction file for X11.

update commands. The following lines depict an outline of a simple instruction file.

```
FROM:    source-path
TO:      destination-path
PACK:    pack-name
p : c1 ... : ... : cn ...
```

One path name *p* is specified in a PACK *pack-name*. It is associated with an update rule consisting of update commands *c1* to *cn*. Suppose the associated update rule is to reproduce the source then the file or subtree *source-path/p* would be copied to *destination-path/p*.

The update process of *Beam* comprises three phases. The first phase parses the instruction file and performs variable substitutions and other preprocessing. In the second phase *Beam* constructs an association of path names with update actions. Path names are subject to file name generation and other mechanisms. The update commands are processed and yield the update action. In the last phase then *Beam* traverses the source and the destination tree simultaneously and performs for each path the registered update action. The update action is executed and creates an internal representation of the destination as it should be. If the actual destination deviates from the internal representation then it is updated. Only the deviating properties are corrected (e.g., only access modes).

Any update rule falls into one of the following four categories. These categories and their effect on a subtree *p* of the software package are explained below. Files, that is, non-directories, are also viewed as subtrees consisting only of a single leaf.

syn This class of update rules is the synthesis of the destination subtree. The simplest form of synthesis is to create an exact copy of the subtree *p* and to keep it up-to-date. More complicated synthesis may include the modification of ownership, access modes, or the contents of files. Subsequently, some update commands will be presented which achieve these modifications.

delete The subtree *p* is not propagated to the destination. If the destination contains a subtree named *p* it is removed. This class of update rules is useful if certain files shall not be distributed because they contain confidential or private information (e.g., license data).

net Instead of copying the subtree *p* to the destination a symbolic link is created referring to the source subtree. Instead of the source path known to *Beam* the user can specify other paths to be used for creating the symbolic links to the network version. This can be useful if the source trees are accessed via a temporary mount point and the net links shall go a different path. This class of update rules is useful if files shall not consume local disk space but must be still accessible.

keep The subtree *p* at the destination side is not updated and remains as it is. Nothing is done if it doesn't exist.

Of course the user can specify update rules for subtrees in *p* pertaining to a different update category than the one registered with *p*. The user can then easily express situations like: link p to the server (net) but update *p/q* (syn) and delete *p/q/r*. The directory *p* and the subtree *q* except *p/q/r* are updated. The siblings of *q* become symbolic links referring to the corresponding file in the source tree. The subtree *p/q/r* is not contained in the destination at all. Arbitrary update rules can be nested in any depth. The nesting of update rules allows the user to express update situations in a very compact way. The PACK *std* in Figure 1 depicts repeated nesting of update rules pertaining to different categories.

Up to now we have implicitly assumed that *Beam* updates a software package from a single source. But *Beam* can handle several source trees. A possible application could be to maintain an original software tree and one which contains only customized files. *Beam* can merge them to a single destination tree. The destination is kept consistent with both source trees. There are update commands which control the selection of source files and their processing.

Beam Instruction Files

All information necessary to perform an update are extracted from an *Beam* instruction file and the arguments passed to *Beam*. An instruction file consists of several sections. Each section starts with a label placed at the beginning of a line. *Beam* parses the contents of sections similarly to a shell. The contents are split into words delimited by whitespace. The quotations ´...´, "...", and `...` can be used in the same fashion as in a Bourne shell [5].

Additionally, Perl [6] code can be placed in an instruction file before the first section label (see Figure 1 lines 1-8). The Perl variables and subroutines can be referred to from the subsequent sections. Macros for update commands or rules can be written in Perl. Furthermore, many update commands allow the specification of Perl subroutines to adapt their behavior to the user's needs. The instruction file depicted in Figure 1 contains several examples of referred Perl variables and subroutines. For instance, the *dst* subroutine in lines 3/4 computes the correct destination path because it differs among the workstations. Simple problems often can be solved by a few lines of Perl code. The setting of Perl variables also can be controlled from the command line by passing arguments of the form: *var=value*. Figure 2 shows an invocation of *Beam*. The Perl variables *src* and *x11* are set from the command line and override the default settings established in the embedded Perl code (Figure 1 line 1/2).

In the following the sections of an instruction file and their meaning are explained in more detail:

FROM: the paths to the source trees

TO: the path to the destination tree

NOTIFY: how to notify users about update events; *Beam* offers three possibilities: sending mail, writing a log file, or using the syslog facility. Each of them can be separately enabled or disabled. *Beam* maps update events to the following priority levels. The following priority levels are associated with update events in increasing importance: messages, updates, warnings and errors. Each of the three notification possibilities can be configured to handle only messages of certain priority. Figure 1 shows a sample setup (lines 11-13): a log file contains all update, warning, and error messages; Warnings and errors are also sent by mail and errors are additionally logged by syslog. The subject line of the mail and the log file name can be set up by printf-like format strings.

PACK: a collection of files and their update rules; a section of this type can occur more than once. PACKs are distinguished by a name which immediately follows the section label. Each line is divided by colons into groups of words. The first group describes a set of paths and the remaining ones specify update commands. The first word of a command group identifies the command and the rest of the words are treated as arguments.

The PACK concept contributes crucially to *Beam*'s flexibility. A PACK describes how a set of files is treated during update. The files are usually related to a certain feature of a software package. PACKs can be included from other PACKs. They also can be referenced from the command line. At our department we use the following naming convention: a PACK name starting with a plus character provides for the feature being copied onto the local disk while those starting with a minus prevent the copying. Instead of copying the files related to a feature symbolic links are inserted referring to the server version. Such PACKs are called mixins while all others are basic versions which can be modified by mixins. Customizing of a software package to the needs of a user reduces to composing a list of PACK names. The user can create instant combinations of PACK on the command line as shown in figure 2. The PACKs *std*, *–misc–fonts*, and *+cc–kit* are composed. Often used combinations can be offered in turn as a PACK. Figure 1 contains a PACK *light-cc* (line 38) defining the same combination of PACK as the instant combination in Figure 2.

In order to facilitate the description of sets of files *Beam* offers a file name generation mechanism and the setting of current working paths. File name generation is a superset of the one available in the csh. The wildcard elements and their meaning are:

- ?, *, [..], [^..] from glob and the shells
- {..,..} as known from csh
- ** matches arbitrary subpaths including the empty subpath
- ^*x* evaluates to all file names in the current directory which do not match the pattern *x*. An initial '^'-sign can be matched by '[^]'.

The pattern **/src/**/^*.[ch] gives an impression of the power of the file name generation. It matches all path names containing a component named *src* and whose last component does not end with '*.c*' or '*.h*' (see also Figure 1 line 16 for another example). In rare cases shell like patterns are inconvenient or not powerful enough. In these cases the user can switch to Perl regular expressions. For instance, Sun patch files are easier to describe with Perl regular expression as with a shell pattern: '.+\.\d{6}–\d\d'.

Setting current working paths eliminates the need to write all path names relative to the root directory of the software package. Once a working path has been established path names can be expressed relative to the working path. Working paths can be stacked. They are set up by adding a final slash to a path name and are removed by solitary slashes. If a working path matches more than one directory the subsequent path names are evaluated relative to each of these directories.

Update Commands

Complex update rules are constructed by combining update commands. For each of the four categories syn, delete, net and keep there is an update command with the same name. This command implements the default behavior as described previously. The default behavior of the *syn* command can be modified by prefixing specific update commands. The rest of this section presents some of these update commands. If the user has not prefixed special update commands *syn* will make the destination an exact copy of the source. If the source is a file tree it is traversed and the update rule is applied recursively.

fgen controls the details of the file name generation. This command can switch to Perl regular expression. File name generation can be

```
% beam -f X11  x11=X11R6 src=/net/future/X11R6  std -misc-fonts +cc-kit
```

Figure 2: sample invocation of *Beam* referring to the instruction file depicted in Figure 1.

restricted to certain types of files, or to certain file trees. The file pattern in the example below will be viewed as a Perl regular expression. The pattern will be only matched against directories. Additionally, a perl function can be specified which further restricts the evaluated set of file names (see Figure 1 line 18).

```
\w+(\.very)?\.old : fgen regexp type=D : syn
```

select is only of interest if there is more than one source tree specified. The default behavior of *Beam* in absence of any explicit selection is to merge the entries of all corresponding source directories. If a file exists in more than one source tree the first one is selected. The order is determined by the order of the source paths in the FROM section. The selection of source files can be restricted to a specific source tree or to a subset of the source trees. The example below illustrates the use of select. The whole subtree *lib* is updated from the first source tree except the libraries *libXYZ** are taken from second.

```
lib/           : select 1 : syn
       libXYZ* : select 2 : syn
```

follow is only active if the source file is a symbolic link. Not the link itself is taken as source but the file it points to. The user can specify a positive or a negative number which means to skip the first n links or to skip n links backward from the end of the link chain. If no argument is given all links are skipped. Additionally, the decision to follow a link can be made dependent on the link contents matching a user specified set of patterns. The example below will skip all those symbolic links as long as they start with either */tmp_mnt* or */amd*. This prevents the update process to copy the symbolic links which have been created by amd(8) or automount(8). Instead the files *behind* these links are copied.

```
* : follow /tmp_mnt /amd : syn
```

translink is only active if the selected source file is a symbolic link. The contents of the link are transformed according to the supplied arguments. The user can specify pairs of strings which represent beginnings and replacements. The contents of the link is matched against all beginnings. If a beginning matches it is substituted by the corresponding replacement. The user can name a Perl array and/or pass some pairs as arguments. Additionally, the name of Perl subroutine can be specified which performs even more complex transformations. This command is necessary if a software package contains links into itself or to its environment and the installation path or the environment at the destination site differs from the source site. Example:

```
@Tb = ("/usr/X11", "/local/X11",
       "/usr/local/bin", "/local/bin");
PACK: ...
  lib man : translink table=Tb /h /home : syn
```

update determines whether the contents of a regular file are viewed as up-to-date by three possibilities of checking: comparing modification times, comparing the contents, and calling a user defined shell command. *Update* applies only to regular files. For an example see the next command.

contents constructs the contents of regular files in case they are viewed as out-of-date. Usually the contents of the selected source file are just copied to the destination file. If more than one source has been selected all selected files are concatenated and written to the destination. Besides these two built-in facilities the user can specify shell commands to construct the contents. The example below shows the commands *update* and *contents* being customized with shell commands. The usage of a *Beam* macro is also depicted. A macro is essentially nothing different from a Perl subroutine which returns a string. The string is substituted for the macro.

```
MACRO cat+sort {
  local($x)='cat $*|sort -u';
  "update   sh=´$x | diff - \$DST´ : ".
  "contents sh=´$x >\$DST´ : syn"; }
PACK: ...
  etc/conf : cat+sort
  bin/*    : contents sh=´cp $1 $DST; \
                      strip $DST´ : syn
```

ino controls the setting of the following attributes stored in the inode: uid, gid, access modes, access time, and modification time. For each of these attributes one of the following operation modes can be chosen: force a certain setting; modify the setting according to some expression (access modes); do not update at all; copy the setting from the source file. The user and group identifications can be either specified by number or by name. The access and modification times can be set to those of some reference file. The example below changes the ownership of all files in the subtree *src* while the group ID is left unchanged.

```
src : ino uid=eirich gid=- : syn
```

Beam still has more commands but they are less often needed than the presented ones. Due to brevity, we only list some of the topics of the omitted commands: sharing of files via hard or symbolic links; testing if files are the target of symbolic links; executing shell commands depending on update events which occurred in a subtree; handling device inodes etc. Complex update rules can be constructed by combining these commands. Because commands are sensitive to the file type an update rule may behave differently depending on the type of the processed file.

An Update Concept for a Cluster of Workstations

An important task in maintaining a cluster of workstations is the keeping of the UNIX installations and replicated software packages on local disks up-to-date. The situation is complicated by the fact that the network is not homogeneous. Often there are several architectures with different kernel architectures. Furthermore, the hardware added to some workstations may also require special software (e.g., a graphic accelerator requires special libraries). Finally, the preferences for software packages may vary from user to user and hence from workstation to workstation (e.g., openwin vs. X11, frame vs. tex).

We have implemented an automated update concept for such a situation using *Beam*. We present the automated update concept as set up in our department. The update includes the UNIX installations of our Sun workstations and several public domain and third party software packages. To simplify the handling of heterogeneity caused by different architectures (sun3, sun4) and different kernel architectures the UNIX installation has been divided into separate trees according to the following criteria:

- machine architecture: this is the main tree. It defines the structure of the UNIX installation and contains most of the software.
- kernel architecture: this tree is related to a specific machine architecture and contains programs and files dependent on a certain kernel architecture (e.g., *top*, *ps*, *vmunix*, kernel debugger etc.)
- local configuration: this tree is architecture independent and contains files describing the configuration of the workstation cluster. It contains files like */etc/amd.map*, */etc/sendmail.cf* etc.

The file trees introduced above are combined by *Beam* to form the UNIX installation of a certain workstation. These file trees are kept on a file server which is accessible by all workstations. System administrators keep the file trees on the server up-to-date by adding, removing, or patching files. Changes are propagated during the night to the workstations. *Beam* instruction files also exist for all other major software packages.

Simply copying software packages in their entirety would fill up the local disks of workstations very quickly. Therefore, *Beam* instruction files offer PACKs to exclude files related to unused features. These features are still available because symbolic links to the server are inserted but they do not require disk space. Inserting symbolic links has a twofold effect. First, other users than the workstation's owner do not miss parts of any software. They are available but probably not on the local disk. Second, if the workstation is operated off the net only those software is on the local disk which is matched by the usage pattern of the owner. Local disks are not clogged with rarely or never used software. A reasonable stand-alone operation is

```
MAILTO eirich                  # mail occasional output of beam-update to ...
M='mail=i4admin'
#     as-user    instructions  parameters   packs
B     -           SUNOS413     $M           std -sunview -rfs -plot -fortran ...
B     -           X11R5        $M           std +misc-fonts
B     eirich      FRAME        mail=eirich  std +english -french
B     -           BIN          $M           eirich
B     src         LEMACS       mail=eirich  std -bytecomp +vm +perl-mode
B     src         PERL         $M           std -curseperl +include -man
```

Figure 3: A sample instruction file of *beam-update*.

	UNIX installation (sun4m)	X11R5	Openwin	Frame Maker 3.1	Lemacs	Sum of all Packages
Total Package (sun4 only)	101MB	68MB	127MB	32MB	31MB	344MB
Reduced (stand-alone)	41MB	16MB	44MB	18MB	15MB	134MB
Ratio (reduced/total)	41%	23%	35%	56%	48%	39%

Table 1: Comparison of full server versions for sun4 architecture with reduced workstation versions. The workstation can be operated stand-alone with the reduced versions.

possible even with a local disk space of only 300MB. Table 1 shows the effect on the size of a software package if barely used parts are omitted. All packages work properly if the workstation is operated stand-alone. The reduced size is compared the a version of the package which already does not contain files related to other than the workstation's architecture.

Up to this point the local disk of a workstation can be filled and updated but a concept for administration and automation of the updates itself is still missing. The automated update uses the UNIX cron service and the program *beam–update*. A workstation destined for an automated update has to run a cron job calling the program *beam–update*. This program determines the host name of the machine it is running on and reads the file */.../beam/update/hostname*. */...* represents the path to the local *Beam* installation. This file specifies the software which shall be *beamed* to the workstation. Figure 3 depicts an instruction file. Each line starting with B describes one *Beam* run. The cron job must be setup under root otherwise it has not enough permissions to update files in the UNIX installation. For certain software packages root permission is not necessary. Therefore *beam–update* allows to specify a user name under who's ID a single *Beam* run is performed. Each *Beam* run is defined by an instruction file, options to *Beam* and a list of PACKs. *Beam* instruction files are implicitly searched in */.../beam/scripts*. The list of PACKs and the options describe the exact configuration of the software packages on the local disk of a workstation.

All relevant *Beam* files for the automated update are located in the file tree of the *Beam* installation which is shared across all workstations via NFS. Thus, maintaining the update process for all workstations is easy. The only thing that has to be done on the workstation itself is setting up a cron job. The update is performed by the clients and proceeds in parallel. Unfortunately, updates are not possible at all if the *Beam* installation is not available, e.g., because the file server hosting the installation is down. This effect can be turned into a more graceful degradation by updating the *Beam* installation itself to the workstations. In this case only updates for those software packages will fail whose server is down.

Current Work

The actual version of *Beam* has some limitations which will be overcome by the work which is currently in progress. The current version of *Beam* relies on the UNIX file system to read and write file trees. If software is not available on some network file system update is not possible at all. This is not very restrictive in workstation clusters but limits *Beam*'s general use. Furthermore, NFS access as root sometimes causes problems due to restricted access privileges. Lastly, if updates of the same software package run simultaneously on several workstations there is some synergetic effect caused by the UNIX buffer cache. Inodes and file data are only read once from disk and are then kept in the cache. But this effect depends on the synchrony of the clients and on the load of the file server host. This synergy could be more efficient if file access to software is controlled by a separate program which does the caching of information relevant to the update process.

The next version of *Beam* will be able do perform updates across a network connection. A daemon *beamd* handles accesses to the file system of the remote host. *Beamd* can be used to read remote software packages (pull file model) or to write updates to a remote host (push file model). Additionally, the new version allows the remote triggering of updates. A software server triggers a client to run an update. The details of the update are completely determined by the client. A configuration file for a *beamd* on each host defines what software is advertised for update and what is allowed to be remotely updated by whom. Different authentication schemes prevent unauthorized access. Even *Beam* instruction files can be read via a connection to a *beamd*. These enhancements are completely transparent to *Beam* instruction files. The software accessible across a network connection to a *beamd* is integrated into *Beam* instruction files by using a special initial path syllable '/beam'. The next two syllables describe the host and the software package. This is similar to the */net* or */amd* mechanisms with the difference that the */beam* path syllable is only valid within *Beam* scripts.

Related Work

There are three other systems that have goals similar to *Beam*: *rdist* [1], *depot* [2, 3] and *track* [4]. *Rdist* uses a push file model while the two others employ a pull file model. Both *rdist* and *track* perform updates across some network connection and do not need a network file system. Hence, they lack the idea of inserting symbolic links to a server version. This fact makes them inappropriate for the presented update situations. They are also quite inflexible because they do not have a mechanism comparable to the PACK concept in *Beam*.

Though the primary intention of *depot* is the management of software environments and not software update some features of *depot* come close to some of the ideas in *Beam*. Depot merges file trees by either copying files from some depot or by creating symbolic links into the depot. The number of commands and their combinations are limited compared to *Beam*. Further, *depot* cannot handle operating system files.

Conclusion

Beam has been in use for about one year at our department and it has proven to be very useful. It has been developed because software update tools currently did not match our needs. *Beam*'s power lies in its flexibility. It can operate on several source trees. Features of software packages can be described by PACKs and customization of updates reduces to combining a list of PACK names. *Beam* offers a great number of update commands to control the details of updating files. And finally, the user can add Perl code for further customization of details of update commands and for an individual configuration of instruction files.

Availability

Beam is available via anonymous ftp from ftp.uni–erlangen.de as /pub/beam/beam.tar.gz .

Author Information

Thomas Eirich studied computer science at the Friedrich-Alexander University of Erlangen-Nürnberg. He received his masters degree in Computer Science in 1989. Since then he is a PhD student at the department of operating systems IMMD IV. His main research interests are distributed, object-oriented operating systems.

References

[1] Cooper, M.A.: *Overhauling Rdist for the '90s*. Proceedings of the Sixth Systems Administration Conference (LISA VI), 1992, pp.175–188

[2] Colyer, W.; Wong, W.: *Depot: A Tool for Managing Software Environments*. Proceedings of the Sixth Systems Administration Conference (LISA VI), 1992, pp. 151–160

[3] Manheimer, K., et al.: *The Depot: A Framework for Sharing Software Installation Across Organizational und UNIX Platform Boundaries*. Proceedings of the Fourth Systems Administration Conference (LISA IV), 1990, pp. 37–46

[4] Nachbar, D.: *When Network File Systems Aren't Enough: Automatic Software Distribution Revisited*. USENIX Conference Proceedings, Summer 1986, pp. 159–171

[5] Bourne, S. R.: *The UNIX shell. AT&T Bell Laboratories Technical Journal*, vol. 57, no. 6, part 2, pp. 1971–1990, July–August 1978

[6] Wall, L.; Schwartz, R. L.: Programming Perl. O'ReillyAssociates, Inc., 1990