STORE

A System for Third Party Software Installation and Maintenance

Anders Christensen
Arne Henrik Juul
Steinar Bang

Store

Store is a system for installing and maintaining third party software on UNIX computers. The setup of Store makes it easier to maintain and keep updated the same versions of software on all architectures in a network consisting of UNIX machines of more that one type.

If you are just curious about what Store is, as a user of a UNIX system or as a system manager considering using Store on your system, then you should read Chapter 3 [Overview], page 4. You should also see Section 6.4 [Documentation], page 19 which tells you where information about software packages may be found.

If you are going to install Store on your system you should read Chapter 5 [Installation], page 12, which attempts to give you a step by step instruction of how to install and configure Store itself.

If you are installing third party software under Store, Chapter 6 [Installing Software under Store], page 15, gives you examples of installing common types of software packages from the internet.

This is edition 2.3 of the Store documentation, covering perl-internal version 1.75.

1 License information

We want Store to be used by people and to be of use for them. We want improvements and changes to be merged back into the main Store distribution. The GNU General Public License seems to work well for the GNU suite of programs so we have deceided to release the Store distribution under the terms of this license.

The Store system free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

Store is distributed in the hope that it will be useful, but **without any warranty**; without even the implied warranty of **merchantability** or **fitness for a particular purpose**. See the GNU General Public License for more details.

2 What is STORE?

STORE is a concept for handling third-party programs under Unix.

The goal of STORE is to give administrators and users a framework that:

- provides a clear view on each single application,
- is transparent to users,
- allows handling several versions of one program,
- keeps information on local modifications,
- eases cooperation and sharing of applications between several organizations with different computer usage patterns,
- eases equivalent installation of programs between different hardware/software platforms, and
- lets administrators choose which applications to make available on each machine, so single applications may be added and removed as needed.

To provide these features, STORE is a set of programs, supporting a central concept on how to organize software installation. The central concept is:

- All files belonging to one single application should be kept together in one single directory tree.
- For each collection of machines in organizations in a cooperating STORE environment, there should be *one and only one* such master directory for each application.
- An application may need some architecture dependent files, and some independent files. In STORE, an application has several versions of architecture dependent files, and one common version of each architecture independent file.
- To ensure availability and acceptable access speed, the application packages should be automatically distributed to local copies, enabling automatic propagation of changes in the original version.
- A virtual directory tree with symbolic links weaves together the multitude of packages, creating a "normal" tree for the end-user and ensuring simple use of programs.

3 Introduction to Store

As described in the brief introduction, Store is mostly a set of concepts and conventions on how to organize and install program packages. The visible parts of Store itself consists of:

1. A directory tree structure.
2. A set of utility programs.

We will start by describing the directory tree structure.

3.1 The /store directory tree

The '`/store`' directory tree looks much like a typical '`/local`' or '`/usr`' directory tree on a UNIX system (i.e. there is a '`/store/bin`' directory, a '`/store/lib`' directory, and so on).

```
                                 /store
                                    |
 +----+----+----+-- ... --+---------+-------------------+
 |    |    |    |         |                             |
bin  lib  man  etc       doc                          store
           |                                    +-------+-------------+
          man1                                  |                     |
                                               khym                 Adm
                              +-----+-----+-----+-- ... --+
                              |     |     |     |         |
                             fbm  emacs  gcc   tex       zoo
```

No files reside in the immediate subdirectories of '`/store`'. Instead we find cryptic soft links, like (if we are on a Sun Sparc machine):

```
/store/bin/gcc -> /store/store/khym/gcc/ver-2.3.3/bin/gcc@sun4os4
```

Alternatively, if we are on a DecStation running Ultrix 4.x[1]:

```
/store/bin/gcc -> /store/store/khym/gcc/ver-2.3.3/bin/gcc@dsult4
```

The observant reader may have noticed the system in the files pointed to by the two softlinks, and (correctly) concluded that the directory tree under '`/store/store/khym/gcc`' may contain some interesting material. So let us take a look at this directory tree:

```
                        /store
                           |
                         store/
                           |
           ...--+----------+---------+---------+---...
                |          |         |         |
              lise1/      khym/     ild/     storlind/
                           |
                          gcc/
                           |
    +------------+---------+-+-----------+-------------------+
    |            |           |           |                   |
registration ver-2.3.3/ src-2.3.3/ src-2.3.3-sun4os4/ src-2.3.3-dsult4/
                 |
                 |
        +--------+---+------------+
        |            |            |
       bin/         lib/        emacs/
        |                         |
    gcc@sun4os4                info/
    gcc@dsult4                   |
```

[1] It will not look *exactly* like this, of course.

```
                                     gcc.info
                                     ...
```

What this directory shows us is that no actual files "live" in the '/store' sub-directories. Instead we have soft links pointing to files collected in *applications*. The name of the '/store' tree used in the Store documentation is *linktree* or *server linktree*, and conceptually, the *macro view*. The directory tree where the files actually reside is called a package's *version tree* or the *micro view*.

The scheme gives us some advantages: First, we don't get any leftover files on deinstallation[2], always a problem when upgrading packages.

Another advantage is that we can have several versions of a package compiled at any given time and install/deinstall different versions with a few simple commands. We can also have softlinks pointing to packages on remote systems, saving space on our local system (but decreasing performance and reliability). By looking at the softlinks, it is immediately apparent to an experienced Store user what application files belongs to, what version of the application, and where to go to find all the files belonging to the application. Even better, this procudure may be automated, so less sophisticated users may find their way around. (NOTE: I have planned to write this automated script, but I haven't gotten around to it yet).

An *application* (or *package*) is a directory containing:

- A 'registration' file holding information about the installed software.
- One or more 'ver-<version>' trees holding a subset of the '/store' tree. All platform-specific files are suffixed with a tag identifying the platform. Files without an identifying tag are common for all architectures.
- One or more 'src-<version>' trees holding the unmodified code.
- One or more 'src-<version>-<platform>' shadow directories, one for each *platform*. The shadow directory holds changed files (changed either manually, or by the patch program), and softlinks pointing to files in the original source tree. This allows us to record and extract changes necessary to make a program (or set of programs) run on a particular platform. It also allows us to keep compilation-ready source trees for a particular package in a space-efficient manner.

In the level above khym in the figure, the other Store file servers are mounted on the machine khym (with khym appearing on the same level in their Store directory trees).

Store also supports *shadowing* of applications. If we take a look at a typical application (in this case gcc, the GNU C compiler):

```
                              /store
                                 |
                               store/
                                 |
     ...--+----------+---------+-+--------------------------+---...
          |          |         |                            |
        lise1/      khym/     ild/                      storlind/
                     |                                      |
                    gcc/                                  .gcc/
                     |                                      |
          +----------+-+------------+-...            +------+-----+
          |            |            |                |            |
      registration ver-2.3.3/  src-2.3.3/       registration ver-2.3.3/
                       |                                          |
                       |                                          |
              +--------+-...                             +--------+-...
              |                                          |
             bin/                                       bin/
              |                                          |
          gcc@sun4os4                               gcc@sun4os4
```

[2] The normal Store deinstallation process would remove all links pointing to a particular package.

```
gcc@sun3os4                                  gcc@sun3os4
gcc@dsult4                                   ...
...
```

The file server **khym** holds the *master* installation of the application gcc. Any changes to the current version, or installation of new versions of the package should take place here.

The file server **storlind** holds a *slave* of the gcc package. The slave copy consists of the 'registration' file and a copy of the 'ver-<version>' directory tree holding the installed files of the package.

Because **khym** holds the master installation it needs to hold files for all architectures using this particular system. In this example, **storlind** is a file server for a site using Sun 4 and Sun 3 machines, which means that only files for these arhictectures, as well as those common for all architectures (e.g. emacs info files) are copied over.

The updating of slave installation of packages is done nightly. The reason for using slave packages is to reduce cross-system NFS access (a STORE system may use a wide area network).

3.2 The main utility programs.

The main utility programs are those that do batch processing or the STORE trees, checking consistency, propagating changes, and updating linktrees. The basic functionality of these has been unchanged since the first versions. See Chapter 8 [Output guide], page 42.

3.2.1 cmaster utility

cmaster runs on each machine having a repository. It checks all the locally defined repositories and finds master applications. It reads their registration files, checks the application for consistency (potentially producing warnings that need to be dealt with), and writes back the registration file. Any output from cmaster probably indicates a real problem with the application.

3.2.2 cslave utility

cslave finds all slave applications on the local STORE server, and fetches changes or new versions from the master servers. Since the master copies of applications will change with time (if the master copy is properly maintained and bugs are fixed!), some output is inevitable. Also, spurious errors may indicate NFS problems or incorrect mounting of filesystems. It is hard at first to recognize which warnings from cslave need manual intervention, but running it manually a couple of times should give some clues. Since it fixes most of the transient problems on the first run, any warnings or errors on the second run may indicate a "permanent" problem, especially if you also make sure that all the repositories involved have been checked with cmaster first.

3.2.3 cclient utility

cclient first processes the linktree to find links that are obviously wrong (like pointing at nonexistant files), or extra non-symlinks. Then, it finds all available applications for the linktree and makes the right symlinks in the linktree, optionally running any "nightly commands" defined by the applications.

3.2.4 report utility

The report utility finds all applications and makes a nice report on them, including a graphic presentation of architectures supported and special files. Most of the information reported is extracted from the registration file directly.

3.2.5 status utility (planned)

The status utility is even more administrator-oriented than the report utility. It will show the amount of disk space occupied by different applications, what types of documentation is available, what source code and build trees are left around, etc. It is intended to help prevent "bit rot" in the STORE system.

3.2.6 autocomp utility

Autocomp is a tool designed to make things easier for people updating applications. It will compile and install applications for different architectures automatically(and simultaneously). Of course, if compiling or installing errors occur, autocomp will fail. (A mail will be sent to notify the one who started it.) Autocomp tries to pick the least loaded host for compilation, and makes sure only one autocomp process is installing in a linktree at a time. It depends on rsh/rcp access from the host autocomp is started to all the hosts that will be used for the actual compilation.

4 Concepts and conventions of Store

When discussing Store, there are many new terms and concepts. We have tried to define the most important concepts here.

4.1 The central repository concept

A *repository* is a directory containing application packages suitable for Store. It must be available as `/store/store/<name>`. You may have several repositories on a single repository server. The naming conventions we use is that repositories are named after the machines.

In the config files, the programs and earlier versions of this documentation the term that was and is used is just "store". This may lead to unfortunate confusion between the term when used for the Store system itself, or when used for the repository concept.

4.2 The macro view

The linktree, also called `/store`, also called the macro view, is what the users see available under the `/store` directory, not including anything under `/store/store`.

4.3 Machines having one or several repositories

A Store server is any machine having a repository or a linktree. In most cases, you would have just one repository and one linktree. If you get short on disk space, and add another disk, using it as an extra repository mostly for compiling is easy. You would only want extra linktrees in a heterogenous environment.

4.4 The application concept

An application, or package, or software, or a program, is whatever is packaged under a `/store/store/<master>/<app>` directory. It is wise to keep a single program released as one package as a Store application. It is generally a very bad idea to have several masters for the same application.

4.5 Versions of applications

You may have several versions of an application in the master repository, with different *maturity levels*. Different version can have different architecture support, so you can compile new versions for just some architectures at first. All slave copies will mimic the master, copying any old or new versions, and deleting versions when they are deleted on the master. Link trees will choose the *best* version according to their level profile and what support the differing versions have. Keeping old versions around may also provide backup, or be useful references when compiling new versions.

4.6 Master repositories

A master repository is a repository containing some masters for applications. In principle, any repository will be checked for masters, but in practice, it is wise to limit the number of master repositories. Master repositories should be on trusted, stable machines, and mounted on most other repository servers to ensure correct copying to slaves.

4.7 Slave repositories

A slave repository is a repository with no master versions, only slave copies of applications. You would probably have several slave repositories for each master repository, but as always, this depends on available disk space. Slave copies only hold the version trees for an application, not source code or build trees, and you may limit the amount of files copied by limiting the number of architectures. You only need to make the slave directory `/store/store/<slave>/.<app>` to make the next nightly run slave down the application, but you may prefer to use `slaveapp` and `linkup`.

4.8 Compile Servers

A compile server is a machine used for compiling applications in STORE for some architecture. It would probably have a local master or slave repository, but it only needs read-write access to a repository to use for compile trees, and read access to the master servers for the applications it should compile.

4.9 Compile trees

A new concept recently introduced in STORE that wasn't originally in the design is the compile tree. This is a bit of a kludge, with some hackery in the tools to support it. But it's very useful.

A master server for an application holds the master registration file, the source code, additional files (doit scripts etc), and probably some shadow trees created with shadow. In the original design, the master should hold shadow trees for all architectures the compilation was compiled for, e.g. sun3, sun4os4, and sgi. The compile server would just mount the repository containing the application read-write, and compile there.

In practice, this isn't feasible. The master server would need to export its repository read-write to a large (and variable) group of compile servers, and compiling on a disk mounted remotely may be nearly impossible. Instead, the compile server creates a pseudo-application for the exact purpose of compiling. If the master is '`mast`', the compile server '`compi`', and the application is '`someapp`' version '`1.23`', the compile tree would be the pseudo-application '`/store/store/compi/someapp-c`'.

The `-c` suffix denotes that this is indeed a compile tree. The shadow utility recognizes this, and looks around in '`/store/store/*/someapp`' for original source code (in this case, '`/store/store/mast/someapp/src-1.23`'). The shadow utility makes a shadow tree on the compile server, with links to the NFS-mounted original source. After compilation and successful installation into '`/store/store/compi/someapp-c/ver-1.23`', the files which differ between the compile server and the master server's version trees should be exactly the architecture-dependant files. These can be `tar`'ed together and transferred to the master server to complete the installation for this specific architecture. (Postinst will try to do this `tar` automatically).

4.10 Generalized architectures

In STORE, an architecture is a description of a set of computers. It may range from any machine to one very specific machine. The name "architecture" should indicate the common meaning, but the concept is deeper than this. For example, "sparc" and "sgi" is two different architectures, but "sparc" may be further subdivided into "sun4os4" or even into "sun4mos413p", indicating a SparcStation 10 series running SunOS 4.1.3 with patches.

For files, architectures are denoted by suffixes using an `@` as a special separator. Mostly, we are only interested in binary files having support for a special hardware/software platform. However, it is sometimes necessary to have configuration files that depend on other attributes of machines, for example mail programs that need to know the local address. The architecture concept, therefore, has been expanded to include other attribute-value pairs. The currently defined attributes are:

architecture – `arch:<value>` or only `<value>`

domain – `domain:<value>` or `d:<value>`

So a file for SparcStations running SunOS 4 could be '`file@sun4os4`', while a file for machines at the mathematics department could be '`file@domain:imf.unit.no`'. In the pathological case of a file with compiled-in configuration, one would need '`file@arch:sun4os4,domain:imf.unit.no`'. This general syntax would also permit additional attributes to be defined, but then some of the internals of the STORE scripts would have to be changed, too.

4.11 The administrative area

The home area for the "store" user contains configuration files for STORE, logs, reports, and may also have special programs to avoid peculiar troubles on special systems.

It is probably a good idea to share the administrative area between all machines running STORE in an administrative domain. This may be done by making '`/store/store/Adm`' a symlink to '`/store/store/<primary>/Adm`', where '`<primary>`' is the main repository server in the administrative domain. The problem is, of course, that if the primary server is down, the directory is unavailable, and you cannot run the STORE scripts, not even manually. However, it makes for much easier administration, as you otherwise would have to keep configuration files up-to-date on several servers.

Dot files: As all users, the "store" user should have a good setup for the shell in use, such as .cshrc/.login or .bashrc/.bash_profile files which are distributed in the inistore package. The most important thing about these files is setting the PATH variable properly. This must be done individually on different systems, since machines, OSes and sites differ substantially in their proper setup. The distributed set will try to source files in a common area, as this is the most widespread method (and also one of the best) for the setup of the shell. In addition, the distributed files will source files provided as part of the STORE scripts, to make STORE support commands available. These are implemented as perl scripts, and directly available as aliases in the shell. (See '`/store/etc/internal/aliases.{csh,bash}`' for details).

Configuration files are contained in the '`etc/`' directory. The main configuration file is just the '`config`' file. See Section 7.2 [etc/config], page 31.

The '`initials`' is part of the `perl-internal` application, and is a site-wide file contains initials for the persons installing and maintaining applications in STORE, with mail address, organization, etc. The primary repository maintainers should update this in the same way as the '`genarchs`' file whenever new staff is hired. See Section 7.3 [genarchs], page 36.

There are also descriptions of the different licensing terms that applies to the programs you put into STORE, and the classifications of programs that you use. These will also be maintained in the same way as the '`initials`' file.

The '`logs/`' directory contains logs from the nightly runs of the STORE programs, and may also be used for other jobs that are run via the *nightly commands* feature.

The '`bin/`' directory is used mainly for holding the nightly script '`night.sh`' that runs the STORE updates. This needs to be modified to be able to: Have a correct `PATH`, start the STORE updates (commonly via `rsh` or `remsh`), and send mail about the result. If you need to make script versions of common programs to avoid errors or deficiencies in your operating system, this is also the place to put such scripts, so they will be found by the STORE programs.

4.12 Cooperating administrative domains

An administrative domain is a set of repository servers administrated together, with a single configuration file. They would probably be located nearby each other net-wise, so it is possible to use any repository from any linktree inside the administrative domain. One could define the administrative domain as the set of machines running the STORE scripts on the same nightly run.

The set of administrative domains NFS-mounting directories and slaving applications from each other is a STORE **site**. The site's **primary repository** is the master for the `perl-internal` application. You don't need to mount every other repository directly - you can slave applications indirectly.

4.13 Different nightly scripts

The Store nightly job is run on the primary server for an administrative domain. It will start the cmaster and cslave scripts on all repository servers in the domain, and the cclient script on all machines hosting a link tree, by using the '`rsh`' command. In addition, it will run the report script on the primary server, to make a report on all available programs. Any nightly jobs specified in the registration files will be run (by `cclient`) for each linktree, with the `TOPDIR` environment variable set to the top of the linktree.

4.14 Using persons' initials for reporting

To make the reports produced by Store more readable, initials should be registered in the registration files. The standard report script will work well with two- to four-letter acronyms, while longer names will be truncated to four letters.

5 Installing Store

This chapter has instructions on how to set up a Store system. It mostly assumes you are setting up a brand new Store system on a site where Store hasn't been used before, but is also useful if you're setting up a new Store administrative domain, and are going to cooperate with other administrators in your organization. In this case, you should probably get some of the other administrators to help you out. In particular, if you're installing Store somewhere at the University of Trondheim, please contact us first!

5.1 Requirements for installing Store

Short checklist of preliminaries:

- A unix system with long filenames, symlinks, and a normal unix command set.
- NFS, and some knowledge of setting up NFS exporting and mounting.
- The perl scripting language.
- Disk space: At least 300MB, recommended 1GB.
- Mounting the disk as '`/store`'.
- Making a Store user, giving it ownership of '`/store`'.

If you are in doubt on these, read the next section carefully!

5.2 Description of preliminary actions

This is a quite long description of the preliminary actions to take before you can really start installing Store, explaining in detail the 'Needed Items' list of requirements .

The Store program system has some special requirements that must be dealt with before installing the Store system itself. The most troublesome requirement is that Store really needs a decent bit of disk space. If you are installing Store on a large installation with lots of old applications, you will probably want to keep the old applications running while you install Store and alternative versions of your applications there. If you don't already have some space free, the advised course is buying a disk before installing Store. A 300MB partition is good for trying out Store, while a full installation with some architectures would call for at least 1GB on the main server.

After getting some disk space, you need to make it available under '`/store`' on all the machines that should run Store. First, let's look at a standalone system. In this case, make a big partition and mount it as '`/store`'. (Or alternately, make a symlink from '`/store`' to another partition - say '`/usr/local/store`' - that has some free space).

You also need to decide which user should own '`/store`' and run the Store scripts. The recommended way is to make a specific "store" user and a "store" group. Currently at the University of Trondheim, this user is uid 52 and the group gid 52, and the "store" user should run "bash" or "tcsh" as shell. Using csh or ksh as equivalents for these should also work, and if you're starting a new, separate Store installation, you can choose other id's if necessary.

The home directory for the "store" user should be taken to be the location of the administrative area for Store, the '`/store/store/Adm`' directory. See Section 4.11 [the administrative area], page 10.

At this point, you are ready for running the initial Store setup script (as the "store" user).

Extra requirements when setting up a differing cases of repository servers: In the common case, you need to export the '`/store`' directory and possibly any '`/store/store/<server>`' directories. Store is written to permit readonly mounting everywhere, since exporting readonly to the world is both simple and secure.

The clients need to mount the server's '`/store`' on '`/store`'. If you have a heterogenous setup, for example you have Sun clients running from an SGI server (or conversely), you need an extra linktree for the clients. See Section 7.2.6 [Serving foreign clients], page 35. You should also propagate the "store" user to the clients.

With all the different setups for mounting and exporting disks, and distribution password information, it is impossible to specify exactly what actions you need to take to export the filesystems from the server, mount them on the clients, and make a common "store" user. If you don't know how to do this on your machines already, you probably need to find out anyway, or hire someone who knows how. Giving specific examples here would be of doubtable use, seeing as there are different export file formats (and names), fstab formats (and names), two different automounters with differing setups, and a couple of NIS/YP versions here already.

(NOTE: I will try to include some amd examples, at least. -arnej)

In addition to all of this, you need the perl scripting language. Decent vendors actually ship perl as part of the operating system these days, and most sites quickly install it otherwise. If you haven't got it already, you need to compile and install it yourself, or get a binary version from somewhere. Perl is also needed for running many other system administration tools, so the sooner you get it, the better!

If you find you need perl, we have a package with perl for various platforms available from ftp.unit.no in '/local/store/perl-4.036.tar.Z'. This is also a good example of a Store application.

5.3 The 'inistore' setup script

The Store setup script is named 'inistore' and is available together with the Store scripts themselves from ftp.ntnu.no in '/local/store/inistore-1.80.tar.Z'.

This script will do some sanity checks, and ask you one (possibly two) questions: Are you installing a new primary repository, and if not, what is the nearby primary repository named. See Section 4.12 [administrative domains], page 10.

First, the inistore setup script will check that it's running as the "repository" user, using the 'id' command. If this command is missing, or the output is not recognized as correct, the script will give a warning, and ask if you want to continue.

Then, the script checks that the '/store' directory exists and is owned by the user. If this fails, the script will abort with a fatal error.

The script needs either uname -n or hostname to work, and will use the short form of the hostname as the name for the repository. If only uname is found, the script will create a hostname script in the administrative area.

If you already have the '/store/store/<host>/Adm/' directory, the script will ask you if you want it removed and reinstalled.

The administrative area with initial files is unpacked into /store/store/<host>/Adm for sharing in your admistrative domain. Additionally, /store/store/Adm is made a symlink to /store/store/<primary>/Adm.

The script will copy the perl scripts used by Store into /store/store/<primary>/perl-internal and set this up as a Store application. The '/store/etc/internal' directory will be made, with softlinks pointing to the perl-internal application. You should not need to change the scripts in the perl-internal application. (The internal nature of the scripts are still changing fast, and isn't documented at all, so you would need to know a lot about both Store and perl to do real modifications). However, on the primary repository you need to maintain the files under 'perl-internal/src/cf' (See Section 7.3 [genarchs], page 36).

The script will try to determine a list of directories to use as a PATH when running scripts. It will try some well-known directories, and will also try using the common files given at the previous questions if possible. This is used in an initial version of the 'nightly' shell script in the bin directory.

5.4 Completing the setup

After running the setup script, you still do not have a complete setup. These are the things you may want to do:

- Check & Edit the config file - See Section 7.2 [etc/config], page 31
- Add architectures - See Section 7.3 [genarchs], page 36
- Add 'initials' definitions - see the "initials" section

- Install some applications! Hopefully, you can get some pre-installed from another site running Store, or we can provide you with a tar-file from our collection (See Section 7.6 [Core Applications], page 40). You should also compile some applications yourself, to acquaint you with the process.
- Check & Edit the nightly script - see below
- Edit users' setup files - see below.

5.4.1 Fixing the nightly script

One of the most important things if you want a stable Store system is to ensure that the nightly script (named '`night.sh`' or '`nightly`') runs regularly, and that the output is taken into consideration by the person(s) maintaining your Store.

The nightly script should run the 'batch' update scripts cmaster.pl, cslave.pl, cclient.pl on all machines in your administrative domain, and report.pl and status.pl once for the whole administrative domain.

If your machine is running standalone (your administrative domain is just this machine), the scripts can be started directly. In all other cases, some sort of remote job must be started. The normal case is to run the scripts via the `rsh` or `remsh` commands. This may provoke a need for '`.rhosts`' files in the Store administrative area, and may also need other customization.

Also, the nightly script needs to have a sensible default PATH variable. The setup script will guess a value, but you probably want to change it.

Lastly, the nightly script needs to send its results via mail to the Store maintainers. This must be customized manually - both the list of recipients and the way to invoke the mail program probably needs changing.

One problem with the current nightly script is that installation of new versions of programs produces many warnings about links being changed and files being copied. The simplest way of dealing with this is just by running the nightly script twice and only collecting warnings on the second go, thus getting just the persistent output — presumably errors that need manual fixing. This may be done by running "nomail.sh" first, then "night.sh" afterwards. "nomail.sh" is exactly like "night.sh", but all output comes directly on stdout. Note that if you run it from cron, the output is not saved.

For further help, there are some examples (See Section 7.4 [night.sh], page 38).

5.4.2 Changes to user setup to accomodate Store.

The standard user login should be changed to make users find the software installed under Store. In addition, certain environment variables may need to be set to make programs work correctly when installed in Store.

Changing the PATH to include '`/store/bin`' and '`/store/gnu/bin`' in appropriate places should be self-evident. However, you will probably also need to change the '`XFILESEARCHPATH`' environment variable, since you now have some application-default files in Store, and some in your 'normal' old location. (Here, we have app-defaults directories under '`/usr/lib/X11`', '`/store/lib/X11`', '`/local/lib/X11`', and '`/local/X11/lib/X11`' !)

Some other environment variables that may need proper setting are '`INFOPATH`', '`MANPATH`', '`EMACSLOADPATH`', '`PAGER`' and '`LD_LIBRARY_PATH`'.

6 Installing software under STORE

Ordinary users of the computer systems running STORE should not notice STORE at all in any way, other than new versions of popular program packages becoming available to everybody.

The main "users" of STORE are the people who install and maintain applications. They need to learn handling the peculiarities of installing programs in STORE. The overview of STORE provides an understanding of the goals of the installation, while this part of the documentation is concerned with the details of how to do it.

When you learn to use the tools provided with STORE, installation in STORE may actually become easier than "normal" installation.

The most useful sections may be the examples, especially if you have little experience with STORE. Looking at the examples (See Section 6.5 [Ordinary programs], page 19), you should be able to install most programs.

6.1 Installation goals

You need to compile the application so it will search under '`/store`' for all its data files. Most programs using GNU configure, for example, can be configured correctly, by simply using `configure --prefix=/store`. Alternatively, you may need to modify the Makefile, or a configuration file.

The program should ideally be as self-contained as possible, and should not refer to any nonstandard files outside of STORE. In some cases, programs may be set up to look for configuration files under '`/etc`' or '`/local`' as well as under '`/store`', but this should be noted in the registration file. If the program needs to have another STORE application in order to work at all, this should be noted in the 'dependant on' field in the registration file – then the STORE scripts will check that this other application is available before slaving or linking up the application (NOTE: this is not implemented yet!)

A binary program would be available as `/store/bin/<program>`, but the goal is then to install it on the master repository as

```
/store/store/<master>/<app>/ver-<ver>/bin/<program>@<arch>
```

Data files go under the `/store/lib` or `/store/lib/<app>` directory, and you need to consider whether the data files are architecture-dependant or not.

Manual pages goes into, for example, `/store/man/man1/<program>.1`. See Section 6.4 [Documentation], page 19.

Of course, all these are only our local convention, and it is possible to modify them to suit other environments. However, they seem to work pretty well here.

6.2 Installing programs — checklist

These are the steps to follow when installing an application:

1. Go to the master STORE for the application, or make the new application on a master repository server.
2. Get the sources for the program, and unpack it.
3. Move the sources into a `src-<version>` directory under the application's directory.
4. Make a shadow tree using "shadow" for the master repository server's architecture.
5. Go to the new `src-<version>-<architecture>` tree. Read any installation instructions for the application.
6. Use "unshadow" or "fix", and make necessary changes to Makefiles, configuration files, etc. to make the program fit under '`/store`'.
7. Compile the program.
8. Either install the program into the `ver-<version>` tree, or
   - install the program directly under '`/store`', and
   - run "postinst" to move the program into the `ver-<version>` tree.

9. Go to the `ver-<version>` directory. Check that all files have a correct architecture prefix.
10. Check the application consistency by running `chkapp -a <app>`.
11. Register the application, or update the registration, using the "register" tool.
12. To test the new version, use "linkup" to ensure that the master's linktree is updated. You should now be able to use the program from the master's linktree. Remember to test that it works for ordinary users, too, not just for the "repository" user!

Furthermore, you should follow these steps when compiling for additional architectures:

1. Go to a compile server for the architecture wanted.
2. Make the compile tree top directory `/store/store/<compi>/<application>-c`.
3. Make a shadow tree by running "shadow".
4. Go to the new `src-<version>-<arch>` directory.
5. As above, modify as necessary, possibly using patches generated when compiling for the main architecture.
6. Compile the program as normal.
7. Install into the `ver-<version>` subtree in the compilation tree, possibly by `make install` and `postinst`.
8. If necessary, rename any architecture-specific files to indicate the correct subset they support.
9. Transfer all architecture-specific files to the master repository for the application. In normal cases, you would use postinst and the tar-file generated by postinst. If you use FTP to transfer the files, make sure the new files have correct permissions. You should have the exact same set of architecture-independent files in the compile tree as in the master's version tree, so you should not need to transfer these.
10. Update the registration file on the master with `register`.
11. Use "slaveapp" to transfer the new version to the compile server's slave repository, and "linkup" to correct the compile server's linktree. Then verify that the programs are correctly available and functional under '`/store`'.

6.3 Tools used when installing

When using STORE, we have found that a need for some tools for effective installation of application. It is also possible to make more such tools.

All these tools are currently implemented as aliases or functions in the shell, mostly just starting a perl script. They could also be implemented as shell scripts installed in the administrative area, in the `bin/` directory, if you use a shell without aliases or equivalent functionality. For example, the "shadow" command is just an alias for "perl /store/etc/internal/shadow.pl".

These are the tools currently available:

- shadow - creates a shadow source tree for compiling packages.
- unshadow - replaces a softlink in the shadow source tree with an ordinary file
- fix - unshadows a link and run the "vi" editor on the file
- postinst - finds programs installed directly under '`/store`' and moves them into the version tree of a package.
- register - creates and updates the '`registration`' file.
- slaveapp - fetches an application from a master repository.
- linkup - creates or updates the softlinks for a program package.
- linkdown - deletes softlinks for a package.
- chkapp - checks that an application is consistent and determines the support.

Common for most of these tools are that they ask cryptic questions with (normally correct) default values, like for example `shadow`:

```
Which compile repository? [khym] ?
Which application? [gopher] ?
What version? [1.03] ?
what architechture [sun4os4] ?
```

The default values are based on the current directory and what repository server you are currently logged onto. If you run the commands from the "correct" place the defaults are correct. You can also give the answers as parameters to the program; the example above is equivalent to:

```
shadow -s khym -a gopher -v 1.03 -A sun4os4
```

This feature is used mostly in scripts.

A short overview of the current behaviour of the different programs are:

6.3.1 "shadow"

This is used for compilation. First, you need to have the source code in the directory `/store/store/<master>/<application>/src-<version>`. For compile trees, you also need to have created the directory `/store/store/<compilerepository>/<application>-c`. Asks for:

machine
: Names the repository you are *compiling* in. Defaults to the current repository.

application
: Names the software package you are creating a shadow tree for. The default value will be correct if you are standing in the right directory.

master repository
: If you are in a compile tree (no `src-<version` directory), `shadow` tries to locate a master repository with source code, and asks you to confirm that it guessed right. If it can't find anything, you may need to NFS-mount the repository.

version
: The version of the package you are installing.

architecture
: The architecture for which you are creating a shadow tree.

In addition to creating the source shadow tree, `shadow` protects the original source code. This is so diffs can be made more easily (if needed, automatically).

Sometimes you want to make a set of patches to be used for all compilations regardless of architectures. In this case, run `shadow` with an architecture of "local", and make your modification in the `src-<local>` shadow tree. On later runs, `shadow` will make links to this new tree instead of the real original, so you get a two-tier system of links.

6.3.2 "unshadow" and "fix"

`unshadow` replaces a softlink with a copy of the file it points to, and ensures that the file copy is writable by the owner (repository). It is just a small shell function, and takes the name of the link as parameter.

`fix` is exactly like `unshadow`, but in addition invokes the "vi" editor on the file, for small modifications.

Note that "unshadow" is *not* the inverse of "shadow".

6.3.3 "postinst"

`postinst` is quite a 'dirty' tool. It assumes that you are installing a complex application, and do not want to unneccessarily modify the Makefiles etc. In this case, it is convenient to do 'make install' into the '`/store`' directory directly. Normally, this would be dead wrong, and the next run of the nightly job with `cclient` would notice all the extra files and delete them.

`postinst` is used right after installing directly into '`/store`', and searches the '`/store`' linktree for extra files, assumes they really belonged to the application you are installing, and moves them into the version tree.

To alleviate some of the worst problems with postinst, it has a built-in time limit: It assumes that files older than 10 minutes doesn't belong with this installation, and leaves them alone. To modify this time limit, use the `-t minutes` option.

In addition, if the application only installs files into a specific directory, you may optimize `postinst` by using the `-p prefix` option. For example, use `-p /store/gnu` when installing GNU programs configured with `configure --prefix=/store/gnu`.

Postinst asks the following extra questions:

architecture
: (option -A) gives the identifying suffix of architecture-dependent files. Postinst uses the "file" program to guess what files are architecture-dependent, and it may guess wrong!

server
: Specifies where the compile tree resides.

application
: Specifies what package you are doing.

version
: Specifies the new version number.

6.3.4 "register"

Creates or updates the file controlling the installation of the application. This program asks many questions. Most questions are hopefully self explanatory. Register has options for evaluating some information itself, which may be useful. Some of the questions are particularly cryptic:

Nightly command
: A command run regularily to update certain files of an installation. Should normally be left empty.

Not links
: Asks for a `perl` regular expression identifying files in the link tree that are *not* softlinks to files in the version tree. This can be temporary files or automatically generated configuration files (e.g. generated by the **Nightly command**). Should normally be left empty.

You may also edit the registration file manually, if you are just updating some fields. Note that if the registration file syntax is wrong, important information may be lost. See Section 7.1 [Registration file], page 29.

6.3.5 "slaveapp"

This tool fetches an application from the master repository. It is almost equal to a subset of `cslave`. It asks for slave repository, application, and master repository. These may also be specified using the `-s`, `-a` and `-m` options.

6.3.6 "linkup"

This tool links the version tree into the link tree ('`/store/bin`', '`/store/lib`' and so on). It is almost equal to a subset of `cclient`. The program reads the '`registration`' file to get most of its information, like what version to link up. It asks for three things:

linktree
: Specifies which link tree you are linking up against. Normally it is the same as for the local machine unless you have separate link trees for clients.

server
: The repository the application is residing in.

application
: The package you are linking up.

6.3.7 "linkdown"

Performs the reverse procedure of linkup — determines what links belong to an application, and deletes them. This program may be used to deinstall an application (temporarily or permanently), and may be useful when installing a new version of some application, since some application are confused by the softlinks in the linktree pointing to the old application version. If you give linkdown the **-F** option, it will use a brute-force search of the linktree and find all files pointing into the application. This may be useful to discover and get rid of old links, or when an application has no registration file to guide normal linkdown (such as when the application is just a compile tree). It asks the same questions as linkup above.

6.4 Handling documentation

To make it easy for users to find documentation, as much as possible should be installed under the `/store/doc/<app>` directory. This means user guides, reference cards, background information etc.

If possible, such documentation should be available as ascii text, dvi files, and postscript files.

Most programs will have at least some manual pages, to be made available under the `/store/man` directory. The directories are divided as follows:

- /store/man/man1 - User programs
- /store/man/man2 - System calls
- /store/man/man3 - Library functions
- /store/man/man4 - Device drivers and network interfaces
- /store/man/man5 - File formats
- /store/man/man6 - Games and Demos
- /store/man/man7 - Misc. information
- /store/man/man8 - Daemons and system administration commands

Note that this follows the traditional BSD layout. In most cases, installing the `nroff` source files will be sufficient. In some cases, it may be desirable to process the manual pages into postscript and put them with other documentation under `/store/doc/<app>` as above.

Most applications from FSF (the Free Software Foundation), like gcc and bison, come with their documentation in the TeXinfo format. The TeXinfo format can be used to create printable documentation (using Don Knuth's TeX typesetting program) and to create info files. Info files are hypertext documents that can be read by Emacs, and by standalone info readers like info, xinfo, and ivinfo.

To process a texinfo file, use the `makeinfo` and `texi2dvi` programs (these are in the texinfo application). In addition, you need to create a one line description to be merged into the top level menu ('`/store/emacs/info/dir`'). The STORE convention for this is to put the description into a file named '`programname.dir.type`' where '`type`' is one of the following:

emacs documentation for Emacs itself

elisp documentation on emacs lisp

packages emacs lisp packages like `calc`

compilers `gcc`, `gdb` and so on

library function libraries (mostly c or c++)

program standalone programs like `tar`

The division into types is controlled by the file '`/store/emacs/info/dir.pre`'. To get a correct syntax for the short-description file, just copy one of the preexisting files — it will be merged into an emacs-info menu.

6.5 Example installation: Gopher

Gopher is a protocol for simple exhange of information. The Gopher package contains several programs (e.g. a server program and a client program). As an example of installation of an ordinary (text oriented) UNIX program we install the Gopher client[1].

We start by installing the source code on the file server `khym`.:

[1] The other programs only becomes interesting when you wish to create your own Gopher server, which is not a suitable program package to install under STORE

```
khym:/store/store/khym$ mkdir gopher
khym:/store/store/khym$ cd gopher
khym:/store/store/khym/gopher$ ftp ftp.ntnu.no
```

Fetch the source code, then check that it is OK to unpack here:

```
khym:/store/store/khym/gopher$ zcat gopher1.03.tar.Z | tar tvf -
rwxr-xr-x1147/10      0 Aug 31 23:47 1992 gopher1.03/
rw-r--r--1147/10   1876 Jun 25 02:14 1992 gopher1.03/README
rwxr-xr-x1147/10      0 Sep  2 23:07 1992 gopher1.03/gopherd/
...
```

This looks good, so we do an actual unpack:

```
khym:/store/store/khym/gopher$ zcat gopher1.03.tar.Z | tar xvf -
... (lots of output)
```

Change the name of the top level directory:

```
khym:/store/store/khym/gopher$ mv gopher1.03 src-1.03
```

Trash the tar-file, which we don't need any longer:

```
khym:/store/store/khym/gopher$ rm gopher1.03.tar.Z
```

Then we create the build tree for Sun Sparc machines on the file server `khym`. The tool `shadow` should be smart enough that we can accept the default values it presents:

```
khym:/store/store/khym/gopher$ shadow
Which machine? [khym] ?
Which application? [gopher] ?
What version? [1.03] ?
What architechture(s) [sparc] ?
Protecting original source in src-1.03
Shadowing from ../src-1.03 to src-1.03-sparc
/store/store/khym/gopher/src-1.03/gopherd:
/store/store/khym/gopher/src-1.03/gopher:
        [...]
/store/store/khym/gopher/src-1.03/examples/Sample Directory:
/store/store/khym/gopher/src-1.03/examples/Sample Directory/.cap:
/store/store/khym/gopher/src-1.03/examples/Sample Directory/wais-index:
Making version tree ver-1.03
```

Then we come to the *tricky* part of the installation: You have to read the '`README`' and the '`INSTALL`' files of the source code and configure the application appropriately. It is difficult to give any general guidelines here, since almost every program has its own unique configuration scheme.

In this example we have to edit '`Makefile.config`' (the simplest way to do it is by issuing the command '`fix Makefile.config`') and then issue the command '`make client`'. In '`Makefile.config`' we had to change the directories '`/store/bin`' and '`/store/lib`' instead of '`/usr/local/bin`' and '`/usr/local/lib`'. In addition we will have to add the location of the local gopher-server.

On Sun machines we need to make sure that all networking applications are loaded with the argument `-lresolv`. This ensures that the applications use DNS to look up machines.

Following these modification the compilation should be straightforward. This brings us to the next stage in the "storification process": the Installation.

This program consists of so few files that it is simplest to do the intallation by hand. The first thing we do is to issue a `make -n` command[2]:

```
khym:/store/store/khym/gopher/src-1.03-sparc/gopher$ make -n install
        install -c gopher /store/bin
        install -c gopher.hlp /store/lib
```

This looks OK, so we do:

[2] Tells make to print what it will do following a given make rule, but *not* to actually do it.

```
khym:/store/store/khym/gopher/src-1.03-sparc/gopher$
     mkdir ../../ver-1.03/bin
khym:/store/store/khym/gopher/src-1.03-sparc/gopher$
     mkdir ../../ver-1.03/lib
khym:/store/store/khym/gopher/src-1.03-sparc/gopher$
     cp gopher ../../ver-1.03/bin/gopher@sparc
khym:/store/store/khym/gopher/src-1.03-sparc/gopher$
     cp gopher.hlp ../../ver-1.03/lib/gopher.hlp
```

The manual page can be found in the 'doc/' catalogue of the source code. We install that as well:

```
khym:/store/store/khym/gopher/src-1.03-sparc/doc$
      mkdir ../../ver-1.03/man
khym:/store/store/khym/gopher/src-1.03-sparc/doc$
      mkdir ../../ver-1.03/man/man1
khym:/store/store/khym/gopher/src-1.03-sparc/doc$
     cp gopher.1 ../../ver-1.03/man/man1/gopher.1
```

The Sun Sparc installation of the gopher program is complete. However, we wish to install this application for the SGI platform as well. Since most servers are exported read only to other machines we can't normally write to the '/store/store/khym/gopher' directory when compiling on another machine. Therefore, the Store system supports shadow directories on other machines than the main installation server for a particular package:

```
flory:/store/store/flory (405)$ mkdir gopher-c
flory:/store/store/flory (406)$ cd gopher-c
flory:/store/store/flory/gopher-c (407)$ shadow
Which machine? [flory] ?
Which application? [gopher-c] ?
What version? [1.03] ?
Using server on khym, app name gopher
What architechture(s) [sgi] ?
Shadowing from /store/store/khym/gopher/src-1.03 to src-1.03-sgi
/store/store/khym/gopher/src-1.03/gopherd:
/store/store/khym/gopher/src-1.03/gopher:
/store/store/khym/gopher/src-1.03/object:
  [...]
/store/store/khym/gopher/src-1.03/examples/Sample Directory/.cap:
/store/store/khym/gopher/src-1.03/examples/Sample Directory/wais-index:
Making version tree ver-1.03
flory:/store/store/flory/gopher-c (408)$ cd src-1.03-sgi/
flory:/store/store/flory/gopher-c/src-1.03-sgi (409)$ fix Makefile.config
```

From here on, the going is easy. We can start out with the changes we did to make the application compile on the Sun architecture, and, if necessary, add flags for the SGI platform. Then we compile the application.

After completing the compile, log in on the machine khym again and copy the gopher application compiled for SGI:

```
khym:/store/store/khym/gopher/ver-1.03/bin$
      cp /store/store/flory/gopher-c/src-1.03-sgi/gopher/gopher gopher@sgi
```

The only thing left is updating the registration of the newly installed package. Here, to illustrate, only thing actually entered by the user is a lot of <return>s to accept the default, and a log entry.

```
khym:/store/store/khym/gopher$ register
What repository [khym] ?
What application [gopher] ?
(Info) (register) <gopher@khym> Updating registration.
(Info) (register) <gopher@khym> Consistency check - be patient.
```

```
Full name . ... ... [Gopher] ?

Available versions are: 1.03 2.012 - choose primary:
Primary Version ... [1.03] ?
Program Type .. ... (? for list) [FO] ?
License Type .. ... (? for list) [Fri] ?

You need to decide what release levels the different versions are.
The different versions support:
1.03 / dsult4/sgi/sun4os4
2.012 / sun4os4
Possible release levels include alpha, beta, gamma,
stable, old, obsolete. 'prealpha' is always last chosen.
Release level for version 1.03 [release] ?
Release level for version 2.012 [alpha] ?
OK, releaselevels are: 1.03/release 2.012/alpha

Signature . ... ... [AHJ] ?
Short Description . (max 30 chars) [Gopher information service] ?
   ==>                              <==
Online Help ... ... [some, in /store/lib/gopher.hlp] ?
Importance  ... ... [3] ?
Source  ... ... ... [boombox.micro.umn.edu] ?
Nightly Command ... [] ?
Not Links . ... ... [] ?
Dependencies .. ... [] ?
Compile Info .. ... [] ?
Current description is:
: Used to access gopher information servers. May also be used as
: an interface to X.500, archie, anon.ftp, etc. You need good local servers
: with well-structured, up-to-date information to fully realize
: gopher's potential.
Do you want to edit the description [N]?
Enter text to be appended (terminate with '.')
Compiled 2.012 for sun4os4. Still the same type configuration.
.
```

We can wait until the nightly commands are run for the links to be created, or if we are impatient to test the installation we can use `linkup`:

```
khym:/store/store/khym/gopher$ linkup
Which linktree? [khym] ?
Which server? [khym] ?
Which application? [gopher] ?
Missing link to ++/khym/gopher/ver-1.03/bin/gopher@sparc
Missing link to ++/khym/gopher/ver-1.03/lib/gopher.hlp
Missing link to ++/khym/gopher/ver-1.03/man/man1/gopher.1
done
```

6.6 Example installation: Xgopher

Xgopher is another gopher client. It is used as an example of how to install a typical X11 application (using `imake` for configuration.

We start by installing the source code (fetching, checking directories and finally unpacking the software):

```
khym:/store/store/khym$ mkdir xgopher
khym:/store/store/khym$ cd xgopher
khym:/store/store/khym/xgopher$ ftp ftp.ntnu.no
```

```
...
khym:/store/store/khym/gopher$ zcat xgopher.1.2.tar.Z _ tar tvf -
  [...]
khym:/store/store/khym/gopher$ zcat xgopher.1.2.tar.Z _ tar xvf -
  [...]
khym:/store/store/khym/gopher$ mv xgopher.1.2 src-1.2
khym:/store/store/khym/gopher$ rm xgopher.1.2.tar.Z
```

We then create the build tree:

```
khym:/store/store/khym/xgopher$ shadow
Which machine? [khym] ?
Which application? [xgopher] ?
What version? [1.2] ?
What architechture(s) [sparc] ?
Protecting original source in src-1.2
Shadowing from ../src-1.2 to src-1.2-sparc
../src-1.2/bitmaps:
Making version tree ver-1.2
```

We now have to do the necessary changes to make the application compile on a Sparc platform. It this case we see from the installation documentation that we need to do some changes in the '`Imakefile`' and the file '`conf.h`'.

We also need to change the applications defaults file and the help file to use Store. Many X applications have an '`app-defaults`' file so it is worth noting that this file must but put into the '`/store/lib/X11/app-defaults`' directory, so that the application will be able to find it. If you wish to change the default choices of colors, fonts and so on, you can do modifications to the effect in the installed application defaults file

```
/store/store/khym/xgopher/ver-1.2/lib/X11/app-defaults/Xgopher
```

For the X libraries to be able to find it at this location (rather than in the default hardwired directory), we need to set the environment variable XFILESEARCHPATH correctly (this is done in the default X setup for the site). The location of the help-file is hardwired into the xgopher application, so this just needs to be set correctly in the '`Imakefile`'.

The changes in the '`conf.h`' file are mainly gopher setup information which we will not touch in this example.

After changing '`Imakefile`' and '`conf.h`' the commend sequence `xmkmf; make` will do most of the work.

In this example as well we will do most of the installation manually (from the directory '`/store/store/khym/xgopher/src-1.2-sparc`':

```
$ make -n install
if [ -d /local/X11/bin ]; then set +x; else (set -x; /bin/sh
/local/X11/bin/mkdirhier /local/X11/bin); fi
install -c   xgopher /local/X11/bin
if [ -d /store/lib ]; then set +x; else (set -x; /bin/sh
/local/X11/bin/mkdirhier /store/lib); fi
install -c -m 0444 xgopher.help /store/lib
if [ -d /store/lib/X11/app-defaults ]; then set +x; else (set -x; /bin/sh
/local/X11/bin/mkdirhier /store/lib/X11/app-defaults); fi
install -c -m 0444 Xgopher.ad /store/lib/X11/app-defaults/Xgopher
if [ -d /store/lib/X11/app-defaults ]; then set +x; else (set -x; /bin/sh
/local/X11/bin/mkdirhier /store/lib/X11/app-defaults); fi
install -c -m 0444 Xgopher-color.ad
/store/lib/X11/app-defaults/Xgopher-color
echo "install in . done"
$ make -n install.man
make -n install.man
if [ -d /local/X11/man/man1 ]; then set +x; else (set -x; /bin/sh
/local/X11/bin/mkdirhier /local/X11/man/man1); fi
```

```
install -c -m 0444 xgopher.man /local/X11/man/man1/xgopher.1
echo "install.man in . done"
$ mkdir ../ver-1.2/bin
$ mkdir ../ver-1.2/lib
$ mkdir ../ver-1.2/lib/X11
$ mkdir ../ver-1.2/lib/X11/app-defaults
$ mkdir ../ver-1.2/man
$ mkdir ../ver-1.2/man/man1
$ cp xgopher ../ver-1.2/bin/xgopher@sparc
$ cp Xgopher.ad ../ver-1.2/lib/X11/app-defaults/Xgopher
$ cp xgopher.man ../ver-1.2/man/man1/xgopher.1
```

In the same way as for the text based gopher client example, xgopher can be compiled for an SGI architecture on the file server `flory`. This examples shows the commands only (output removed in the interest of brevity):

```
/store/store/flory$ mkdir xgopher-c
/store/store/flory$ cd xgopher-c
/store/store/flory/xgopher-c$ shadow
/store/store/flory/xgopher-c$ cd src-1.2-sgi
/store/store/flory/xgopher-c/src-1.2-sgi$ rm Imakefile conf.h
/store/store/flory/xgopher-c/src-1.2-sgi$
      cp /store/store/khym/xgopher/src-1.2-sparc/Imakefile .
/store/store/flory/xgopher-c/src-1.2-sgi$
      cp /store/store/khym/xgopher/src-1.2-sparc/conf.h .
/store/store/flory/xgopher-c/src-1.2-sgi$ vi Imakefile
/store/store/flory/xgopher-c/src-1.2-sgi$ xmkmf
/store/store/flory/xgopher-c/src-1.2-sgi$ make
```

We then skip back to `khym` and do the installation of the executable by hand (that is: copy the file
'`/store/store/flory/xgopher-c/src-1.2-sgi/xgopher`' over to
'`/store/store/khym/xgopher/ver-1.2/bin/xgopher@sgi`'.

We then register xgopher:

```
What repository [khym] ?
What application [xgopher] ?
(Info) (register) <xgopher@khym> Updating registration.
(Info) (register) <xgopher@khym> Consistency check - be patient.

Full name . ... ... [X gopher grensesnitt] ?

Available versions are: 1.2 1.3 1.3.1 - choose primary:
Primary Version ... [1.3.1] ?
Program Type .. ... (? for list) [FO] ?
License Type .. ... (? for list) [Fri] ?

You need to decide what release levels the different versions are.
The different versions support:
1.2 / dsult4/sgi/sun3/sun4os4
1.3 / dsult4/m68k/m88k/sgi/sun4os4
1.3.1 / dsult4/hp700ux9/m68k/m88k/rs6aix32/sgi/sparc
Possible release levels include alpha, beta, gamma,
stable, old, obsolete. 'prealpha' is always last chosen.
Release level for version 1.2 [obsolete] ?
Release level for version 1.3 [old] ?
Release level for version 1.3.1 [release] ?
OK, releaselevels are: 1.2/obsolete 1.3/old 1.3.1/release

Signature . ... ... [TIW] ?
```

```
Short Description . (max 30 chars) [X-klient mot informasjonstj.] ?
   ==>                            <==
Online Help ... ... [litt, trykk '?'] ?
Importance  ... ... [4] ?
Source  ... ... ... [ftp.cso.uiuc.edu:uiuc/src/xgopher.${version}.tar.Z] ?
Nightly Command ... [] ?
Not Links . ... ... [] ?
Dependencies .. ... [] ?
Compile Info .. ... [conf=info,inst=info,make=info] ?
Current description is:
: Nice-looking interface to the gopher services. Quite unusable.
: Actually much better, almost usable, in version 1.2!
: And even more elegant in 1.3 !!
Do you want to edit the description [N]?
Enter text to be appended (terminate with '.')
.
```

Note that in this example version 1.1 had already been installed. The description partially refers to the previous version.

6.7 Installing emacs lisp packages

Emacs is a large program with lots of possibilities. For Emacs to be able to use Store, we have to make some changes to Emacs. Most of the changes are done when installing Emacs, seeing to that it is compiled with the correct load-path, options and X libraries. We must also make sure that the DOC-file is identical on all architecture (or that we use DOC-files with an unique suffix for each type of architecture.

You also need special conventions to handle the installation of emacs-lisp packages. Emacs *should* be compiled with the following load-path: '`/store/elisp /store/emacs/elisp`'. This enusres that we have at least two alternative directories in which to place emacs-lisp code. This gives us the possibility of overriding emacs-defaults by installing files with the same name in '`/store/elisp`'. Usually we are installing completely new packages. In which case we usually install them in '`/store/emacs/lisp`'.

For new packages to be accessible to Emacs, we need to insert code which will be run by Emacs on startup. There are two ways of handling this: the first one is to let users install code in their personal '`~/.emacs`' file. The second is to make code being included in the '`/store/emacs/lisp/default.el`' file.

For an application like e.g. auc-tex, which radically changes the the behaviour from the standard TeX-mode, it's probably best to let individual users install the start-up code themselves. How to do this is well documented in the auc-tex emacs info file. In this sample installation we only install the lisp files under '`.../auc-tex/ver-6.1/emacs/lisp`' and leave it up to the user whether or not to use them.

For other packages it may be more suitable to make the installation common to all users. Since each package needs to have a few lines inserted into '`default.el`', this file is generated automatically from the different emacs-lisp applications' pieces of code. The Store convention is to make each application have a file '`.../app/ver-ver/emacs/lisp/default.el-app`'. These files are then concatednated to create the final '`default.el`' file.

An example: The application "term-lock" is very simple. It consists of a simple emacs lisp file, which let you lock your emacs (on a terminal) using a simple password. The file defines the function `M-x term-lock`. To make it publically available we create the file '`.../ver-90/emacs/lisp/default.el-term-lock`' containing the line:

```
(autoload 'term-lock "term-lock" "Lock the terminal" t)
```

A somewhat more advanced example is the USENET newsreader "GNUS". This newsreader has a lot of files we install in the '`.../ver-3.13/emacs/lisp`' directory. Also GNUS has a couple of functions we wish to publicise by inserting autoload lines as above. We create a '`default.el-gnus`' file with the following contents:

```
(autoload 'gnus "gnus" ""
Read network news." t)
(autoload 'gnus-post-news "gnuspost" ""
Post a news article." t)
(defvar gnus-use-generic-from t)
(defvar gnus-nntp-server "news.ntnu.no")
(defvar gnus-use-generic-path "imf.unit.no")
(defvar gnus-your-domain "imf.unit.no")
(defvar gnus-your-organization "Norwegian Institute of Technology")
```

As you can see from the contents, having the same version of this file for all sites using the same Store system will create undesirable effects. We threfore split the file on the basis of the link tree name, making it specific for each organization using ut. Strictly speaking this gives us an overhead since separate machines in the same orgranization may need separate files. This files will be residing in the same master directory and can thus be easiliy maintained.

The contents of the directory '/store/store/khym/gnus/ver-3.13/emacs/lisp' will look like this:

```
default.el-gnus@chanur    gnus-user-sxa.el     gnuspost.elc
default.el-gnus@flory     gnus-user-tale.el    mhspool.el
default.el-gnus@hitra     gnus.el              mhspool.elc
default.el-gnus@hufsa     gnus.elc             nnspool.el
default.el-gnus@ild       gnusmail.el          nnspool.elc
default.el-gnus@kari      gnusmail.elc         nntp.el
default.el-gnus@khym      gnusmisc.el          nntp.elc
default.el-gnus@lise1     gnusmisc.elc         tcp-not-needed.el
default.el-gnus@tootikki  gnuspost.el          tcp-not-needed.elc
```

Luckily GNUS use a simple ASCII file for the configuration. Applications with the domain name compiled into them are much more difficult to handle.

6.8 Example installation: ange-ftp

A good example of emacs lisp packages and their installation is the ange-ftp package. This package overloads some of the basic emacs commands (such as C-x C-f, find-file) to support getting files with ftp using an extended file name syntax (kinda like VMS).

In this case, the source code consists of just two files: The emacs lisp 'source code', and the texinfo documentation source code. A step-by-step installation goes like this:

1. makeinfo ange-ftp.texinfo
2. texi2dvi ange-ftp.texinfo
3. dvips -f < ange-ftp.dvi > ange-ftp.ps
4. In emacs: M-x byte-compile-file ange-ftp.el
5. Copy ange-ftp.ps and ange-ftp.dvi into the ver-X.XX/doc/ange-ftp directory.
6. Copy ange-ftp.el and ange-ftp.elc into the ver-X.XX/emacs/lisp directory.
7. Copy ange-ftp.info into the ver-X.XX/emacs/info directory.
8. In the same directory, make a file called ange-ftp.dir.packages, containing the following line:

   ```
   * Ange-FTP (ange-ftp.info):     Transparent FTP support for GNU Emacs
   ```

   This will be merged into the 'dir' file as a menu choice, in the subsection Emacs-lisp packages.

Now, users will have to put the following line in their .emacs files to use ange-ftp: `(require 'ange-ftp)` - if ange-ftp is always wanted for all users, this line may be put into a file named default.el-ange-ftp in the ver-X.XX/emacs/lisp directory.

6.9 Example installation: gcc

Store really shines with a lot of small applications. These are really hopeless to keep track of with a usual installation, but easily installed and maintained with Store. However, Store can also manage more complicated programs, like gcc. This means the installer of the programs must be more experienced, but also that others can use a complex program without great pain in installing it.

As of the current version, gcc is quite easy to install on the 'supported' architectures: configure with the prefix you wish, in this case /store, and say 'make bootstrap ; make install'. This takes a lot of disk space but is otherwise mostly painless.

The problem with gcc in Store stems from two facts:

1. gcc itself has mechanisms for specifying target architectures and versions.
2. gcc needs a tree of 'fixed' include files that is very different in content between platforms.

Both of these mean a lot of files will be specific for differing architectures, and would need to be marked as 'unneeded' for all other architectures.

Instead of marking all these files with their specific architectures, they are all put in an architecture-dependant directory. The current semantics of such a directory implies that these files are all optional, and that they all are implicitly tagged with the directory's tag. This simplifies the master version tree significantly, but means some extra pain in the installation process. The installation process may, however, be largely automated as long as everything works.

The binary files under the arch-dep directory need not have their own tags, but using postinst during installation, they usually will have their own tags.

The resulting layout of the version tree will then look like this:

```
ver-2.4.5 +-> bin
          +-> man +-> man1
          +-> doc +-> gcc
          +-> emacs +-> info
          +-> lib +-> gcc-lib@alpha +-> alpha-dec-osf +-> 2.4.5
          |       +-> gcc-lib@dsosf1 +-> mips-dec-osf1 +-> 2.4.5
          |       +-> gcc-lib@dsult4 +-> mips-dec-ultrix +-> 2.4.5
          |       +-> gcc-lib@m88k +-> m88k-dolphin-svr3 +-> 2.4.5
          |       +-> gcc-lib@sgi3i4 +-> mips-sgi-irix4 +-> 2.4.5
          |       +-> gcc-lib@sun4os4 +-> sparc-sun-sunos41 +-> 2.4.5
          |       \-> gcc-lib@sun4os5 +-> gcc-lib +-> sparc-sun-solaris2
          |                           \-> sparc-sun-solaris2 +-> 2.4.5
          |
          +-> alpha-dec-osf@alpha +-> include
          +-> m88k-dolphin-svr3@m88k +-> include
          +-> mips-dec-osf1@dsosf1 +-> include
          +-> mips-dec-ultrix@dsult4 +-> include
          +-> mips-sgi-irix4@sgi3i4 +-> include
          +-> sparc-sun-solaris2@sun4os5 +-> include
          \-> sparc-sun-sunos41@sun4os4 +-> include
```

(Actually, there lots of extra directories and include files under the 2.4.5 directories, but that's not relevant to this discussion).

The compilation sequence would look like this, showing '$' only as the prompt:

```
$ mkdir /store/store/khym/gcc
$ cd !$
$ ftp prep.ai.mit.edu
[... get gcc-2.4.5.tar.gz...]
$ zcat gcc-2.4.5.tar.gz | tar xvf -
$ rm gcc-2.4.5.tar.gz
$ mv gcc-2.4.5 src-2.4.5
$ shadow -a gcc -m khym -s khym -A sun4os4
$ cd src-2.4.5-sun4os4
$ ./configure --prefix=/store sparc-sun-sunos41
$ make bootstrap
```

```
$ make install
$ cd ../ver-2.4.5
$ postinst -a gcc -s khym -A sun4os4 -v 2.4.5
$ postinst -t 300 -p /store/lib/gcc-lib
$ rm bin/sparc-sun-sunos41-gcc@sun4os4
$ mv lib/gcc-lib lib/gcc-lib@sun4os4
$ strip lib/gcc-lib@sun4os4/sparc-sun-sunos41/$version/cc1*
$ strip lib/gcc-lib@sun4os4/sparc-sun-sunos41/$version/cpp*
$ mv sparc-sun-sunos41 sparc-sun-sunos41@sun4os4
```

The extra run of postinst is necessary because gcc possibly installs include files with old modification times. This procedure has been evolved through time, and will need changes as new versions of gcc becomes available. The overall concepts remain, however, the same as for simple applications.

7 Example configuration files

This chapter has some example configuration files for repositories, and a detailed explanation of the features available.

7.1 Example registration file

To provide a full example of a complex registration file the GCC application was chosen. Note that there are many fields in addition to the one manually entered with the "register" utility - these are internal to STORE, and updated automatically.

```
Full name . ... ... : GNU C kompilator
Primary Version ... : 2.5.8
Release Levels  ... : 2.4.5/dated 2.5.8/release
Program Type .. ... : PL - Programming languages
License Type .. ... : GPL - General Public License
Signature . ... ... : AHJ
Short Description . : C, C++ and ObjC compiler
Manual Pages .. ... : 3 files - 138 KB
Other Documentation : 1 files - 1716 KB
Emacs Info  ... ... : 30 files - 1194 KB
Online Help ... ... : none
Examples .. ... ... : none
Init Files  ... ... : none
Importance  ... ... : 9
Source Code ... ... : 795-files 23580-kb
Source  ... ... ... : prep.ai.mit.edu
Nightly Command ... :
Not Links . ... ... :
Master pathname ... : /store/store/ludvig/gcc
Master repository .. ... : ludvig
Path to slave . ... : ludvig
Versions .. ... ... : 2.4.5 2.5.8
Locks . ... ... ... :
Compile Info .. ... :
Dependencies .. ... :
Begin(Versionlines)
: 2.4.5 / alpha/m88k/pmax/rs6aix32/sgi3i4/sparc
: 2.5.8 / alpha/linux/m88k/pmax/rs6aix32/sgi3i4/sgi4i5/...
End(Versionlines)
Begin(MasterVlines)
: 2.4.5 / alpha/m88k/pmax/rs6aix32/sgi3i4/sparc
: 2.5.8 / alpha/linux/m88k/pmax/rs6aix32/sgi3i4/sgi4i5/...
End(MasterVlines)
Begin(Description)
: The Free Software Foundation's GNU C compiler is regarded as the best
: general C-compiler around. It produces code for many different
: processors and all major Unix platforms. The C++ part is more buggy
: and has many known problems.
End(Description)
Begin(Log)
: getlogin=arnej, uid=store,  Sun Nov  7 02:42:08 MET 1993
:   Name mangling has changed again, so people using C++ libraries
:   compiled with 2.4 must use 2.4.5.
End(Log)
```

7.1.1 Manually entered information

As seen in the examples for the register utility, these fields are entered by the installer:

Full name The full name of the application (useful to expand acronyms, etc.).

Primary Version The version used to tally files (see below).

Release Levels Each version has a different level, on a scale from newer to older.

Program Type A broad classification of the program. Uses its own configuration file of program types.

License Type Classification of the program's license. Most programs currently in repository are freely available. Also has its own configuration file.

Signature The signature of the person responsible for the program, so people contact this person if there are problems with it, or if the need new versions compiled.

Short Description A quick description of what the program actually does.

Online Help An assessment of the amount of online help available in the program. Seldom used.

Importance On a scale from 1 to 9, how important this program is. Only an indication of how important the installer thought it was.

Source Where the program is available. If possible, use the originating site, so it is easy to check if there are new versions available, even if you probably would fetch it from the nearest FTP mirror.

Nightly Command Programs that need to auto-configure something periodically should have a nightly command. (See Section 7.5 [Nightly commands], page 39).

Not Links Used in conjunction with the previous, when the nightly command generates files in the '`/store`' tree.

Dependencies If the program is dependant on other packages, enter the names of the packages it need. For example, emacs-lisp packages need emacs installed. (NOTE: The information isn't actually used other than manually, by STORE maintainers. Eventually, cslave and cclient will use it automatically).

Compile Info Holds info for the planned autocomp utility.

Locks To signal work-in-progress on an application, different types of locks may be used. Currently the only defined lock is `noupdate`, which means slave repositories should not update from the master automatically. Also note that you cannot enter this field with `register`, but must manually edit the configuration file. This is a deliberate restriction.

Description A longer description used only by the STORE personnel. Mostly used to give a longer description of why the program is useful, tips for compiling, in-jokes etc.

Log Log entries when new version come, problems are fixed, etc.

7.1.2 Computed information

Some of the computed information is only updated when you run `register` - the fields tallying different types of files: **Manual Pages**, **Other Documentation**, **Emacs Info**, **Source Code**, **Examples**, and **Init Files**.

The remaining fields are updated each time you run `cmaster`.

Master repository The master repository for the application.

Master pathname The directory where the master found the application.

Path to slave The repositories the application has been slaved via. Used to prevent slave loops.

Versions Active versions.

Versionlines Architecture support for each version, as seen on the current repository.

MasterVlines The same, but unchanged from the master repository.

7.2 Configuration file

The main STORE configuration file is the administrative area's '`etc/config`' file. There will be one such configuration file for each administrative domain.

7.2.1 Quick guide to etc/config features

Blank lines are ignored. Lines may be continued using backslash-newline. Keywords must be at start of lines.

Keywords:

internal takes a list of repositories in your administrative domains. Links will be made to any repository in this list unless you modify the default metrics.

external takes a list of other known repositories to slave from if available.

store declares a repository. Takes a repository name and *attribute=value* pairs. Required attributes are `host=<host>` (must match the value of '`uname -n`') and `archs=<arch list>` declaring architectures to slave down.

linktree declares a linktree, with linktree name and attribute=value pairs. Requires the `host=<host>` `arch=<arch>` `domain=<domain>` attributes, where the architecture is the most specific name of the host's architecture, and the domain is the most-specific mail or organization domain the host belongs to. Also common is the `unwantedfile=<file>` to exclude applications from consideration.

compile declares where you compile for an architecture, gives sane defaults to `shadow` and `postinst`. Like, `compile <arch> on=<host>`.

report takes a list of up to 10 architectures for `report` to include in the report.out file.

7.2.2 Metrics

There are two types of metrics in the STORE system: From linktrees to repositories, and between different repositories.

Linktree to repository metrics

The linktree-to-store metrics control what repositories a particular linktree may get symlinks to and the preference between these repositories.

For example, if the linktree "flipper" has metrics to the following repositories:

store "flipper"	metric 1
store "flipper-b"	metric 2
store "nova"	metric 5
store "ernie"	metric 150
all other repositories	metric 10000

The limit where symlinks will be created is 100, so this means that the "flipper" linktree wil get symlinks to applications in repositories "flipper", "flipper-b" and "nova" in this prefered order, but never to any other repositories.

Manual linkup will not check metrics or the linkup metric limit (See Section 6.3.6 ["linkup"], page 18).

Store to store metrics

The store-to-store metrics control from where a slave store will get its applications. At the present time there is a limit of 100000 for slaving, so one may be aware of remote repositories without actually copying from them.

For example, if the store "flipper" has metrics

store "nova"	metric 5
store "ernie"	metric 150
all other repositories	metric 10000

This means that, for a slave application in store "flipper", it will first try to slave from the store "nova", then "ernie", then others. Slaving manually can override the metrics and the metric limit (See Section 6.3.5 ["slaveapp"], page 18).

Metric syntax

metric <n> symlink <lkts> to-store <sts>

metric <n> slave <sts> from-store <sts>

where <lkts> is a list of linktrees or possibly the keyword "internal", meaning all linktrees specified in this configuration file, and <sts> is a list of repositories, or "internal", meaning the list of repositories defined as "internal" in the top of the config file, or "external", similarly meaning the list of repositories defined as "external" in the config file.

Default metrics

If you don't specify any metrics the following defaults apply:

Any linktree <l> will get metric 1 to a store <l> with the same name if such a store exists. Any linktree on host <h> will get metric 2 to any repositories that are also on host <h> except where the previous rule applies. All internal linktrees will get metric 5 to all internal repositories except where the previous two rules apply. All internal linktrees will get metric 10000 to all external repositories.

All internal repositories will get metric 5 to all other internal repositories. All internal repositories will get metric 10000 to all external repositories.

The default metrics are designed to fulfill the normal expectations. Unless you're really sure you want to modify them, you should leave the configuration file without any metric specification, so you'll get the defaults.

Metric evaluation and priority

Metrics in the configuration file are evaluated in the order they appear. Earlier specifications in the configuration file will have priority; later specifications of the same metric will be silently ignored.

This means that the general forms with "internal" and "external" will not override any metric that has already been determined, but they **will** override any metric (even a completely specific one) appearing later in the file.

The defaults are applied after evaluating the entire configuration file, using the same rules. So they will not override any user-specified metrics.

This means that, for example, with this config-file extract:

```
metric 5  symlink  flipper  to-store   flipper
metric 7  symlink  flipper  to-store   flipper-b
metric 15 symlink  nova     to-store   flipper
metric 10 symlink  internal from-store internal
metric 20 slave    internal from-store internal
metric 30 slave    internal from-store ernie vier ild
metric 50 slave    internal from-store external
```

The only default metric with any effect applied after this is the one (in effect) specifying

```
metric 10000 symlink internal to-store external
```

In particular, the default rules for linktree/store pairs with the same name or on the same host will not be effective, so the linktree nova will get metric 10 both to the store nova and the store flipper-b, which was probably not intended.

7.2.3 Full walkthrough of the etc/config syntax

If in doubt, consult the source (‘`/store/etc/internal/conffile.pl`’).

The configuration file is parsed line-by-line, with backslash-newline deleted first. Keywords are only recognized at start of lines, and anything else is ignored. There is no comment possibility. *Attribute=value* pairs are whitespace-separated, while lists are comma-separated.

The `top` keyword and the `logdir` attribute are part of a scheme to allow a repository-like system using different names as a separate universe, but using the same scripts. It is not documented at all, and you should not use it. It is only mentioned here for completeness.

Host specifications (with `on=<host>`, `from=<host>` or `host=<host>`) should preferrably match exactly the value returned by the shell command `hostname`, and we recommend that you use the full name. However, only the first component will be compared, because we have found that various OS'es have real problems getting this consistently right.

```
compile <arch> on=<host> [on=<host> ...]

run <what> on=<host> pass=<number> [ from=<host> via=<method> ]

internal <repositoryname> [<repositoryname> ...]

external <repositoryname> [<repositoryname> ...]

top <dir> <dir>

report <arch> <archs> [ <arch> ... ]

store <repositoryname> host=<host> archs=<arch list> \
      [ dir=<dir> ]              \
      [ logdir=<dir> ]           \
      [ domains=<domain list> ] \
      [ levels=<level list> ]   \
      [ autoadd=<repository list> ]  \
      [ unwantedfile=<file> ]

linktree <linktreename> host=<host> arch=<arch> domain=<domain> \
      [ level=<level> ]       \
      [ dir=<dir> ]           \
      [ logdir=<dir> ]        \
      [ unwantedfile=<file> ]
```

7.2.4 Simple config example

Here is the simplest configuration file in actual use at our site today, for the standalone server **ludvig.solan.unit.no** (our primary repository server). It has two disks with repositories, ludvig and ludvig2, for a total of 4GB. It has mostly masters for core application like perl, emacs, X11, etc.

```
internal ludvig ludvig2
external khym ild nova vier lastebil
```

```
store ludvig  host=ludvig archs=sun4os4
store ludvig2 host=ludvig archs=sun4os4

linktree ludvig host=ludvig \
         arch=sun4os412 domain=solan.unit.no \
         unwantedfile=/store/store/Adm/etc/uwfile.ludvig

compile sun4os4 on=ludvig

report sun4os4 none allarchs
```

7.2.5 Complex config example

Here is the most complex configuration file in actual use at our site today, for the administrative domain at **pvv.unit.no**. PVV is the most heterogenous site, having just two Sparc/SunOS machines, one Sparc/Solaris machine, one HP-700, one IBM workstation, one DECstation running DecOSF, a Dolphin m88k server, and a PC running Linux. Each machine has its own repository and its own linktree. The main repository server (nova) has a 2GB repository disk.

```
internal nova dekk bigblue bob datter flipper supernova
external khym ild vier ludvig ludvig2 lastebil \
         flory hani smola grip mari1 kari iptibm3 \
         petra1 taiyang storlind

store nova host=nova archs=sun4os4,rs6aix32,m88k,hp700ux9,linux,sun4os5 \
      autoadd=khym,ild,ludvig,ludvig2,vier,lastebil \
      unwantedfile=/store/store/Adm/uw.pvv
store dekk host=dekk archs=dsosf1
store bigblue host=bigblue archs=rs6aix32
store bob host=bob archs=hp700ux9
store datter host=datter archs=sun4os5
store flipper host=flipper.pvv.unit.no archs=m88k
store supernova host=supernova archs=sun4os4

linktree nova host=nova level=newer arch=sun4os413 domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv level=newer

linktree supernova \
         host=supernova \
         arch=sun4os41B \
         domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv \
         level=newer

linktree bigblue host=bigblue level=newer arch=rs6aix32 domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv

linktree bob host=bob level=newer arch=hp700ux9 domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv

linktree datter host=datter level=newer arch=sun4os5 domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv

linktree dekk host=dekk level=newer arch=dsosf1 domain=pvv.unit.no \
         unwantedfile=/store/store/Adm/uw.pvv

linktree flipper host=flipper.pvv.unit.no \
         level=newer arch=m88k domain=pvv.unit.no \
```

```
          unwantedfile=/store/store/Adm/uw.pvv

run cmaster on=nova pass=1
run cslave  on=nova pass=2
run cclient on=nova pass=3

run cslave  on=flipper.pvv.unit.no pass=3 from=nova via=rsh
run cslave  on=bigblue             pass=3 from=nova via=rsh
run cslave  on=datter              pass=3 from=nova via=rsh
run cslave  on=dekk                pass=3 from=nova via=rsh
run cslave  on=bob                 pass=3 from=nova via=rsh
run cslave  on=supernova           pass=3 from=nova via=rsh

run cclient on=flipper.pvv.unit.no pass=4 from=nova via=rsh
run cclient on=bigblue             pass=4 from=nova via=rsh
run cclient on=datter              pass=4 from=nova via=rsh
run cclient on=dekk                pass=4 from=nova via=rsh
run cclient on=bob                 pass=4 from=nova via=rsh
run cclient on=supernova           pass=4 from=nova via=rsh

run report  on=nova pass=4

compile m88k     on=flipper.pvv.unit.no
compile dsosf1   on=dekk
compile rs6aix32 on=bigblue
compile sun4os5  on=datter
compile hp700ux9 on=bob
compile sun4os4  on=nova on=supernova

report sun4os4 sun4os5 dsosf1 m88k rs6aix32 \
       hp700ux9 linux 386netbsd sgi4i5 sgi3i4
```

7.2.6 A configuration file with foreign linktrees

Now say you have a large disk server with several different types of clients, and you want to make linktrees for the clients on the server, as well as a linktree for the server itself. Now '`/store`' will hold the server's own linktree, '`/store/store/large`' is the actual repository with lots of packages, and you allocate the directories '`/linktrees/sun3`', '`/linktrees/linux`' and '`/linktrees/sun4os5`' to hold linktrees. The names here doesn't really matter, as long as the server exports the directories, and the clients mount the correct linktree directory on their private '`/store`', as well as '`/store/store/large`'.

Let us further assume that some of the sun3 clients really belongs in a maildomain different from the server; they could be at a student lab named *studlab.somewhere* while the rest of the hosts are in the *host.somewhere* domain.

The configuration file could look something like this:

```
internal large
external foo bar baz

store large   host=large.host.somewhere  archs=sun3,sun4os4,sun4os5,linux

linktree large          host=large.host.somewhere \
         arch=sun4os413 domain=host.somewhere \
         dir=/store

linktree largesun3      host=large.host.somewhere \
         arch=sun3      domain=host.somewhere \
         dir=/linktrees/sun3
```

```
linktree largestudlab   host=large.host.somewhere \
         arch=sun3      domain=studlab.somewhere \
         dir=/linktrees/studlab

linktree largesun4os5   host=large.host.somewhere \
         arch=sun4os53  domain=host.somewhere \
         dir=/linktrees/sun4os5

linktree largelinux     host=large.host.somewhere \
         arch=linux     domain=host.somewhere \
         dir=/linktrees/linux

compile sun4os4 on=large.host.somewhere
compile sun3    on=small.host.somewhere
compile linux   on=pc.host.somewhere
compile sun4os5 on=sol.host.somewhere

report sun4os4 sun4os5 sun3 linux
```

The server's '/etc/exports' file could look like:

```
/linktrees/sun3     -access=sun3hosts
/linktrees/sun4os5  -access=sun4os5hosts
/linktrees/linux    -access=linuxhosts
/linktrees/studlab  -access=studlabhosts
/store/store/large  -ro
```

The sun3 clients' '/etc/fstab' could look like this:

```
large.host.somewhere:/linktrees/sun3      /store             nfs hard,ro,intr
large.host.somewhere:/store/store/large   /store/store/large nfs hard,ro,intr
```

Now this is a hypothetical example, but it should hopefully give some good guidelines on how to configure disk servers with foreign linktrees.

7.3 The genarchs configuration file

The genarchs configuration file defines architectures and domains recognized by STORE. You will need to define your site's domains in this file to be able to use domain-specific files. Also, if you have architectures on your site that we don't have here, you need to add them alongside those already defined.

You should only need to edit the genarchs file on your site's primary repository, so this section is really only relevant to the first STORE installation on a site. If someone else is maintaining the primary repository, ask them to add architectures or domains that you need.

The machine with the primary repository will (with the default installation) have a directory with the name '/store/store/<primary>/perl-internal' where <primary> is the name of the primary repository. To edit the genarchs file, follow this checklist:

1. become the "store" user
2. go to the directory '/store/store/<primary>/perl-internal/src/cf'
3. check out with `co -l genarchs`
4. edit with `emacs genarchs` or whatever
5. check in with `ci -u genarchs`
6. go up with `cd ..`
7. do a `make install`

You may want to change the version number in the Makefile first, so you have both the old and the new version running in parallel first.

7.3.1 Adding architectures

Adding an extra architecture should be relatively straight-forward. The relevant part of the configuration file looks like this:

```
# Normal architectures - add architectures as needed:
expand arch    allarchs         bigend litend
expand arch    bigend           sun sgi apollo hp ibm m88k
expand arch    litend           pmax i386 vax alpha
expand arch    sun              sun3 sparc
expand arch    sun3             sun3os4
expand arch    sun3os4          sun3os40 sun3os41 sun3os411
expand arch    sparc            sun4os4 sun4os5
expand arch    sun4os4          sun4os40 sun4os41 sun4os41B
expand arch    sun4os41         sun4os411 sun4os412 sun4os413
expand arch    sun4os5          sun4os50 sun4os51 sun4os52 sun4os53
...
```

To add SunOS 4.1.4 for the sparc platform (fat chance), just change the sun4os41 line to:

```
expand arch    sun4os41         sun4os411 sun4os412 sun4os413 sun4os414
```

For a more drastic example, let's say you start running repository on your Cray machines running UNICOS 6.0, 6.1, and 7.0. Then we need to add another family altogether. The Cray is big-endian, so change the bigend line and add like this:

```
expand arch    bigend           sun sgi apollo hp ibm m88k cray
expand arch    cray             unicos6 unicos7
expand arch    unicos6          unicos60 unicos61
```

7.3.2 Adding and changing domains

The domain part of the genarchs file will probably be more interesting to change, as you will probably want to rip out the part pertaining to the University of Trondheim and substitute your own site. Whether you follow strict mail-domain lines or use some other administrative boundaries as your guide is entirely up to you - do as you feel is most natural. We have found the greatest benefit of domain-specific files to be setup files for mail and news programs, so we follow mail-domains very closely.

The relevant part of the genarchs file looks like this:

```
# Mail/administration domains - add your own domains as needed.
expand domain   alldomains      unit.no sintef.no
expand domain   unit.no         idt.unit.no fm.unit.no pvv.unit.no \
                                stud.unit.no kjemi.unit.no ipt.unit.no \
                                dsl.unit.no ifi.unit.no elkraft.unit.no
expand domain   fm.unit.no      imf.unit.no phys.unit.no
expand domain   phys.unit.no    nobipol.unit.no
expand domain   stud.unit.no    lise.unit.no solan.unit.no siri.unit.no \
                                marin.unit.no alkymi.unit.no petra.unit.no
expand domain   sintef.no       sima.sintef.no runit.sintef.no
```

If we wanted to have a global Store installation, it would probably have to look something like this:

```
expand domain   alldomains      edu com gov mil net org countries
expand domain   edu             mit.edu berkeley.edu toronto.edu \
                                22cf.edu aa.edu ac.edu accd.edu \
                                ...
expand domain   com             sun.com sgi.com dec.com cray.com \
                                10a.com 1776.com 23kgroup.com 24hour.com \
                                24stex.com 2600.com 35sys.com 3com.com \
                                ...
expand domain   countries       no se fi dk is us ca de nl ae ag al an \
                                aq ar arpa at au bb be bf bg bh bm bo \
```

```
                                      ...
expand domain   no                    unit.no uio.no uit.no uib.no sintef.no \
                                      ...
expand domain   unit.no               idt.unit.no fm.unit.no pvv.unit.no \
                                      stud.unit.no kjemi.unit.no ipt.unit.no \
                                      dsl.unit.no ifi.unit.no elkraft.unit.no
...
```

The syntax and semantics should be quite simple, but if you get problems, send a mail and we'll try to help you.

7.4 night.sh script examples

The nightly script must take care of running the nigthly perl jobs `cmaster` and `cslave` on all hosts having repositories, and `cclient` wherever there is a linktree. In addition it should run `report`.

The night.sh script itself is most commonly started from cron. Therefore it may need to set the PATH variable, although the STORE scripts themselves use very few shell commands.

Also, the errors and warnings from the scripts should be logged and (preferably) mailed to the local STORE administrators.

One simple situation is with a standalone machine, where a typical repository script could look like this:

```
PATH=$HOME/bin:/store/bin:/store/gnu/bin:/local/bin:/usr/bin
export PATH
ADM=/store/store/Adm
OUT=$ADM/output
LOG=$ADM/logs/nightly.log
MAIL=mail
PERSONS=root
SCRIPTS=/store/etc/internal

exec >$OUT 2>&1

echo "(Info) Store nightly job start `date`"
perl $SCRIPTS/cmaster.pl
perl $SCRIPTS/cslave.pl
perl $SCRIPTS/cclient.pl
perl $SCRIPTS/report.pl
echo "(Info) Store nightly job stop `date`"

$MAIL -s "Nightly Store problems" $PERSONS < $OUT
cat $OUT >> $LOG
exit 0
```

Using the `run` keyword in your configuration file properly, you can use the `nightly.pl` perl script to actually start the perl scripts remote. Here's a real world example for the same host that has the complex configuration file detailed in this documentation (See Section 7.2.5 [Complex etc/config], page 34):

```
#! /bin/sh
# nightly script, to be run from crontab
#    checks vital and trivial ting about the Store system

ADM=/store/store/Adm
OUT=$ADM/output
MAIL=/usr/ucb/mail
PERSONS="arnej stigb tegge"
SCRIPTS=/store/etc/internal
```

```
PATH=$HOME/bin:/store/bin:/store/gnu/bin:/local/bin:\
/local/gnu/bin:/local/X11/bin:/usr/lang:/usr/ucb:/bin:\
/usr/bin:/local/etc:/etc:/usr/etc:/etc/bin:/local/games:/usr/games
export PATH

exec >$OUT 2>&1

perl $SCRIPTS/nightly.pl

cat $OUT >> $ADM/logs/nightly.log

$MAIL -s "Nightly Store problems" $PERSONS < $OUT
exit 0
```

On another host, where logs aren't as important, the "store" user's crontab file starts the nightly.pl script directly:

```
lastebil:/store/store/lastebil/Adm$ crontab -l
1 1 * * * perl /store/etc/internal/nightly.pl | \
          /usr/sbin/Mail -s "Nightly Store problems" arnej hanche vindvad
```

7.5 Nightly Command examples

The most common way to do nightly commands is to write a small script. So rplay, a sound package with remote sound possibilities, has a registered Nightly Command with the value `perl /store/etc/make-rplay.conf.pl` that is run on each `cclient` run. As is usual in such cases, the purpose of the script is to update a configuration file, so the registered Not Links value is `etc/rplay.conf`, meaning that the perl script should construct /store/etc/rplay.conf (in this case, by finding sound files by searching '`/store/lib/sound`').

Another example is the makewhatis packages. The relevant parts of the registration file looks like this:

```
Nightly Command ... : /store/etc/catman
Not Links . ... ... : man/whatis.* gnu/man/whatis.*
```

The '`/store/etc/catman`' script itself is quite short, so is reproduced here as a good example of how to write a nightly command script:

```
#!/bin/sh

topdir=${TOPDIR-/store}

for MD in $topdir/man $topdir/gnu/man ; do
  if test -d $MD && test -w $MD; then
    rm -rf /tmp/mkw.log.* /tmp/mkw.err.*
    perl /store/etc/makewhatis.pl -M $MD \
      >/tmp/mkw.log.$$ 2>/tmp/mkw.err.$$
    if test -s /tmp/mkw.err.$$; then
      echo "(Warning) Errors when trying to run makewhatis on $MD:"
      sed 's/^/          /' < /tmp/mkw.err.$$
      ( cat /tmp/mkw.log.$$ /tmp/mkw.err.$$ \
        >> /store/store/Adm/logs/makewhatis.log ) 2>/dev/null
      rm -f /tmp/mkw.log.$$ /tmp/mkw.err.$$
    else
      rm -f /tmp/mkw.log.$$ /tmp/mkw.err.$$
    fi
  fi
done
```

7.6 List of core applications

Here's a list of packages that I consider to be some of the most important in repository, as indicated by the fact that they are compiled for most architectures. The report format is as generated by the report utility. The selection is obviously somewhat biased, but I can't list everything: On last count we had more than 500 applications total.

```
P R O G R A M S   I N   S T O R E                                      Page  1

         386netbsd ---+ +--- sgi4i5                              Examples ---+
           dsult4 ---+| |+--- hp700ux9                       Online help ---+|
         sun4os5 ---+|| ||+--- linux             Hardcopy Documentation ---+||
        sun4os4 ---+||| |||+--- rs6000                   Manual pages  ---+|||
     i386sol24 ---+|||| ||||+--- allarchs                 Emacs info  ---+||||
                  ||||| |||||                                            |||||
Name    PrimVer   VVVVV VVVVV Categ. Sign Short description              VVVVV
------------------------------------------------------------------------------
X11        5.26   ***** **-*+ os     TE   MIT Reference Port of X11      -*---
amd        920824 -*+*- ****+ os     AHJ  AutoMounter Daemon             ***--
bash       1.14.3 ***** ****+ os     TE   FSF's bourne shell             ****-
bison      1.21   ***** ****+ pwb    AHJ  GNU version of yacc            ***--
byacc      1.9    -**** *-*-+ pwb    AHJ  Berkeley version of yacc       -*---
c-archie   1.4.1  ***** ***++ info   AHJ  Archie commandline client      -*---
chimera    1.63   -**** ****+ info   AHJ  Web client (X/Athena)          -*-*-
ctwm       3.2p1  ***** ****+ app    MS   Fancy window manager           -*---
detach     1.1    ***** ****+ adm    AHJ  Start programs in background   -*---
dvipsk     5.55a  -**-* **--+ tex    HH-O DVI to PostScript converter    **-*-
elm        2.4.23 ***** **-++ mail   Sig  Interactive mail program       -***-
emacs19    19.28  ***** ****+ edit   HE   The Editor.                    ---*-
fileutils  3.9    ***** ****+ shu    AHJ  Various file utilities         -*---
flex       2.4.6  -**** ****+ pwb    AHJ  GNU version of lex from PWB    **---
fvwm       1.24r  ***** ****+ app    SSB  Motif-like fast virtual WM     -*---
gawk       2.15.2 -**** ***-+ shu    AHJ  GNU version of the awk languag ***--
gcc        2.5.8  ****- ****+ pl     AHJ  C, C++ and ObjC compiler       ***--
gdb        4.13   ****- ****+ pwb    HE   GNU debugger                   **---
gdiff      2.5    ***** ****+ shu    AHJ  difference-computing programs  *----
gfind      3.8    -**** ****+ shu    AHJ  GNU version of find            -*---
ghostscrip 2.6.1. ***** ****+ app    HaNH Postscript interpreter         -**--
ghostview  1.5    -**** ****+ app    HE   Frontend for Ghostscript       -**--
gm4        1.4    ***** ****+ shu    AHJ  Macropreprosessor              *----
gmake      3.69   ***** ****+ pwb    AHJ  GNU make(1) program            *----
gnuplot    3.5    -**** **-*+ app    MS   Interactive plotting program   -*-**
gpatch     2.1    ***** **-*+ pwb    AHJ  Update sources from diffs      -*---
groff      1.08   -**** ****+ os     AHJ  GNU [nt]roff text formatter    -**--
gtar       1.11.2 -***- **-*+ arch   HE   backup/archival tool           *----
gzip       1.2.4  ***** ****+ arch   AHJ  GNU (de)compression - *.gz     ***--
```

```
P R O G R A M S   I N   S T O R E                                    Page  2

         386netbsd ---+ +--- sgi4i5                             Examples ---+
           dsult4 ---+| |+--- hp700ux9                      Online help ---+|
         sun4os5 ---+|| ||+--- linux              Hardcopy Documentation ---+||
        sun4os4 ---+||| |||+--- rs6000                    Manual pages  ---+|||
     i386sol24 ---+|||| ||||+--- allarchs                  Emacs info  ---+||||
                  ||||| |||||                                             |||||
Name    PrimVer   VVVVV VVVVV Categ. Sign Short description               VVVVV
-----------------------------------------------------------------------------
host        930926 ***** ****+ net    AHJ  Simple yet powerful DNS tool   -*---
ispell      3.1.13 -**** **-*+ app    Sig  Spelling checker               **-*-
latex2e     1994.1 ***** ***** tex    HH-O Macro package for TeX          --*--
less        177    ***** ****+ app    HE   Pages, searches, &c            -*-*-
libg++      2.5.2  ****- ****+ lib    AHJ  GNU C++ class library          -*---
lndir       5      ***** ****+ pwb    AHJ  Create shadow tree to sourceco -*---
lynx        2.3    -**** ****+ info   MS   Curses based W3 browser        --***
makewhatis 1990    ***** ***** adm    AHJ  Create whatis databases        -----
mosaic      2.5    ****- ****+ info   SSB  X-based WWW client from NCSA   ---*-
ncftp       1.7.1  ***-* **-*+ net    HH-O Alternative UI for FTP         -*-*-
pbmplus     1mar19 ***** ***-+ ip     MS   Image conversion and manip.    -*---
perl        4.036  ***** ****+ pl     AHJ  Scripting language             -*---
rcs         5.6.0. ***** **-*+ os     SSB  Simple revision control system -*---
rdist       6.1bet -**** **--+ adm    TE   Distribution of files          -*---
shellutils 1.12    ***** ****+ shu    AHJ  Various shell utilities        **---
startxcmd  1.2     ***** ***** shu    AHJ  Start remote X programs        -----
storedoc    2.3    ***** ***** doc    AHJ  Documentation of Store         *-*--
tcsh        6.05p  ***** *-**+ sh     MS   C shell variant                -**--
texinfo     3.1    ***** **-++ tex    AHJ  Emacs-info utilities           ***--
textutils  1.11    -**** ****+ shu    AHJ  Various text utilities         **---
web2c       5.8515 -**** **-*+ tex    HH-O web2c with TeX, MF and utils   -*---
xdvik       1.8    -**** **-++ tex    HH-O Preview DVI files under X11    -*---
xlock       1.08ah -**** ****+ app    AHJ  Lock your screen (for X)       -*---
```

8 Understanding STORE output

This chapter should provide an understanding of the typical nightly STORE job output, and how to fix the problems it indicates.

Store format is formatted according to a schema so it should be easy to parse (both for humans and computers). There is a label (in parantheses) indicating severity (typically one of Error, Warning, Info, or Trace) - normally you only get Info and Trace messages when running a program interactively. An *Error* always indicates something requiring human intervention, while a *Warning* may signify a transient condition. A particular version of an application in a repository is indicated using the schema <*app*@*repository* v*ver*>, so for example version 2.1 of the `zoo` application at repository `khym` is <.zoo@khym v2.1>.

8.0.1 Top 10 errors and warnings

Here are some of the most common output types illustrated with examples and explanations.

1.

```
(Warning) These applications will not be used by khym:
          gindent       nova      (supports sun4os4)
```

Means that there are applications that support the given linktree, but which cannot be used because they aren't slaved to a nearby repository (according to the metrics). You need to decide either:

- The application is something you want, in which case, use `slaveapp` and `linkup` to get it.
- You do not want the application in this linktree, so put it in the linktree's `unwantedfile`.

2.

```
(Warning) Empty symlinks in /store on khym (removed):
 doc/elm ==> Ref.ps Users.ps
```

Symlinks in the given linktree did not point to anything, so they were removed. This may be expected (e.g. because of deinstallation) or it may be because somebody forgot to install files for a new version of some program (the above example is typical).

3.

```
(Error) <traceroute@ludvig> has no registration file
```

The given application is improperly installed - it does not have a registration file. The person responsible for the package should either finish the installation or remove the package if they decide it should no be installed after all.

4.

```
(Warning) bin/gmake
          symlink -> /store/store/ludvig/gmake/ver-3.67/bin/gmake
          symlink -> make
          i.e. nowhere!
```

The application has a symlink that doesn't point anywhere, either because it points in the wrong direction, or because file were removed. In this case, the symlink should be to '`/store/gnu/bin/make`'. Fix the symlink or install the missing files as appropriate.

5.

```
(Error) <latex-nfss@ild> No winning master
```

A slave application could not find any master to slave from. Typically the name of the application is misspelled or the master application has been deleted, in which case you probably want to deinstall the slave as well (using `linkdown` and `rm -rf`).

6.

```
(Warning) Changed link <- /store/store/kari/.elm/ver-2.4.23/man/man1/frm.1
             -> /store/store/kari/fsp/ver-271/man/man1/frm
(Warning) Changed link <- /store/store/kari/fsp/ver-271/man/man1/frm
```

```
                    -> /store/store/kari/.elm/ver-2.4.23/man/man1/frm.1
```

Here, two different application contains the same logical file (a so-called namespace collision). This need to be resolved by renaming on of the files somehow, even if it means reconfiguring and recompiling one of the applications involved.

9 Miscellaneous

This chapter has some background information on Store: The background and history of Store, the authors, the current TODO list, and where to reach us.

9.1 The problem background

Store was created out of the needs that we felt, managing a large, heterogenous network of Unix machines. Despite claims in trade rags, administrating unix isn't trivial. A list of factors create complexity in the total system (different hardware, different OSes, lots of third-party programs, different versions of "everything"). This complexity makes for a messy and unmanageable system. Some of the most common problems a system administrator meets is:

- Directories like `/local/bin`, `/local/lib` and `/local/man/man1` often contains a great mass of files, but nobody really knows what *all* the files do. Nobody dares to remove anything, because it *may* be important.
- A short while after a program is installed, nobody is really sure that it is still consistent, because nobody is sure what files belong to that program.
- Often it is unclear what version of a program that is actually installed, since many programs do not have any means of displaying version or patchlevel. The source code may not be available, or (worse) only the source code for another, older version is available.
- Because of limited disk space, source code is often removed shortly after installing a program. Later, one realizes that a great deal of work had been spent locally on porting and adjusting the program to fit the local machines and setup. Source code is sometimes available on backup, but finding it may be very time-consuming.

  When the next version of the program needs to be installed, the same adjustments must be reconstructed. The same problems appear when one fixes a bug locally that is not merged into the official distribution.
- On the other hand, source code may be lying around for ages because everyone believes it is precious and cannot be regenerated.
- Different people install programs in different ways and use different file system layouts. Even if there are a local strategy, some programs will not fit because they require special treatment. If you want to find all files belonging to a program, you must spend a long time looking through your file system.
- When managing several separate installation and has to separately install programs on all systems, they are often only installed on a subset of the installations, where the installer needs them him/herself, or where the pressure from users is greatest. Thereby some installation must do without some programs, or they are left with old versions.
- To obtain good performance, it is often desirable to make copies of programs on several machines for use from local disk, but administrating this distribution takes a lot of time.
- Making report generators for what programs are available or finding unneeded files is near-impossible,

These are the problems Store is trying to solve.

9.2 The history of Store

The idea and basic concepts behind Store has evolved from the similar "depot" system made by NIST and CMU. The first pilot version was implemented (mostly with shell scripts) and saw some use on the student computing labs.

Later, a new and enhanced prototype version was implemented on behalf of the IMF by Anders Christensen, a student at IDT, as a half-year project (spring of 1992). It was renamed Store so as not to be confused with the NIST/CMU depot system.

At this time, the basic concepts were well-defined. Coming into full use at IMF and Alkymi, many of the more practical tools made possible by Store was written, and needed optimizations was implemented. During the following year, the internals of Store changed a great deal, and attempts was made to translate most of the programs' external interfaces into English.

9.3 About the authors

The original versions of STORE was conceived and implemented by Anders Christensen. Arne Henrik Juul did a lot of the improvements and further hacking as well as writing the examples on how to install software under STORE. Steinar Bang wrote the original version of this TeXinfo document, translating documentation written by the other two and adding some of his own. Also involved with the underlying algorithms has been Tor Egge.

9.4 Todo list

This section holds the current todo list for STORE:

- Update the TODO list.
- Flesh out, clarify, and generally clean up this documentation. Write some more documentation, man-pages, etc.
- Document generic architectures, version levels, and several versions (in progress).
- Finish the `inistore` distribution and installation script for STORE. Document it.
- Make STORE easier to configure.
- Consistency checking of the STORE scripts.
- Better (automatic) notification of newly installed packages. Maybe automatically posted to USENET newsgroups or sent by email.
- Better possibilities to ignore applications, just getting reports when new versions arrive.
- Finish translation of STORE to English. Mostly done.
- Update `proginfo`, a user-oriented script.
- Automate patch generation and auto cleanup of compile trees.
- Creating tools for browsing the entire STORE installation for installed program packages. Maybe making the information available with *gopher* would be a good idea?
- Actually implement the `status` tool, reporting disk usage, changes, etc.
- Precompute disk space requirements and avoid filling up disks. This requires some way of getting the free/total space on the disk, which probably means requiring GNU df.
- Make the scripts more resistent to network problems.
- Support dependencies between applications, so e.g. xdvi can be dependent upon TeX fonts being installed.
- Make postinst more robust by checking a database of notlinks patterns, and implement some link tree locking to avoid several people trying to use postinst at the same time, or someone using postinst as the same time as cclient is deleting files and running nightly commands.
- Better support for setuid/setgid programs. Today, you get a message that something needs to be done, and have to fix these problems manually.
- Fix so cclient checks notlinks and complains if there's just a link, anyway.
- Better support empty directories and directories with special permissions. Today, only files in the version trees are significant, and you need to add additional "nightly jobs" to fix permissions. cclient even deletes empty directories in the link tree.
- Make implementation as database of relations possible, so you don't need the actual symlinks, but can hardlink or copy directly into the (now) link tree.
- Make a track interface and .tar.gz interface for machines wanting to run programs compiled for STORE by just unpacking into their '`/store`' directories (which may be just a symlink to '`/local`').
- Make a track or ftp transport mode, instead of using NFS. This will greatly boost reliability and performance over WANs.
- Investigate the possibilities of implementing a user-level NFS server doing the link tree to actual file conversion.

- Changing the 'linkup' and 'linkdown' tools so that they can be run by the owner of a particular directory. This would mean that people could install STORE packages without knowing the password of the "repository" user (someone with system priviledges would create a package directory and changing the user to the account of the maintainer and the group ownership to the "repository" group).
- An 'install' script that will attempt to infer the installation directory and instead install into the version tree with a suitable suffix. This is to "trick" 'Makefiles' into installing programs directly into the version tree. The 'install' script should guess the correct installation directory from names like '/usr/bin', '/local/bin', and so on, to make it easier to install software without a flexible installation setup.

9.5 Contact persons

For electronic mail contact, try `arnej@lise.unit.no` and if you do not receive a prompt answer, try `storeadm@imf.unit.no` whereby you'll reach a greater group using STORE.

The primary distribution site for STORE is `ftp.unit.no`, our local FTP server. Currently it is put into the `/local/store` directory in the anonymous FTP area.

To reach us by phone, try:

- +47 7359 3535 - ask for Arne H. Juul
- +47 7359 3681 - ask for Anders Christensen

Concept Index

Table of Contents