The FreeBSD ports system.

Michel Talon

August 2006

Abstract

We present the main features of the FreeBSD ports system, explaining how it is structured, how software is installed and updated, and compare it with the Debian system.

Contents

1 Introduction.

The FreeBSD ports system is a complex collection of tools and data files allowing to retreive, compile and install software packages on a computer in a coherent way. One can also install precompiled software packages. Finally there are tools allowing to maintain the installed software in current state. This system has been invented by one of the founders of the FreeBSD project, Jordan Hubbard, and checked in CVS in 1993. Subsequently, J. Hubbard created the installer sysinstall, checked in in 1995, which makes use

of the above system to install packages in the course of endowing the computer with an operating system. In 1997, NetBSD decided to import the FreeBSD system, and provided improvements, some of which have been imported back in FreeBSD, see [1] for a fine discussion of the NetBSD system. Of course FreeBSD developers have also improved their system, so in the course of time, these tools diverged to the point that the present NetBSD system is called pkgsrc, and has several more advanced features, described in [2]. On the other hand the FreeBSD system has vastly more ports than the NetBSD one, as can be seen by browsing the pkgsrc site:
`http://www.netbsd.org/Documentation/software/packages.html`

Anyways, both systems share essentially the same philosophy, and i will discuss only the FreeBSD ports, by lack of sufficient knowledge of pkgsrc. One can get an excellent view of many aspects of pkgsrc by reading the pkgsrc guide [3] which is a model of clarity, like most of NetBSD documentation. A bird's view of pkgsrc shows that bsd.port.mk has been completely rewritten, split into many parts, each having different function, cruft has been removed, etc. For example the task of building a package list is splitted into mk/plist/plist.mk, and incidentally, it has automatic building. The main idea of J. Hubbard has been to use the BSD **make** tool as the central tool organizing the work of package building. Rules are provided in the form of Makefiles, explaining how to fetch, extract, patch, compile and install each port. Since building a package frequently needs to install another one, this logic also allows to recursively build and install all required dependencies. This automatically solves the dependency problem for initial installation of software which marred many Linux distributions at the time this system was introduced. Installing a port consists just in going to the appropriate port directory and typing:

```
make install clean
```

All the rest occurs automatically as long as one has an internet connection to fetch program sources and of course, if no problem happens, such has unfetchable source, program not compiling, etc. Even if a problem occurs it is usually possible to solve it manually, by finding the source somewhere else, hacking the source, etc. A problem which does not appear is the "dll hell", which was so characteristic of the first Linux distributions when installing binary rpm found randomly on the web. So this system has known a wide popularity. However, with the advent of very complicated interdependent software packages like gnome, involving tenths of ports, it appeared that tools to manage such a net of dependencies when upgrading the machine would be useful, in the same vein as the Debian tool apt-get. Such a tool has been introduced by Akinori Musha, in 2001, and is called portupgrade. It consists of several Ruby scripts located in the Ruby lib directory `/usr/local/lib/ruby/site_ruby/1.8` plus several user level tools allowing to get information on the installed packages and automatically upgrade them. This system has been variously appreciated, some swear by it, others say they always end up with a broken machine using it, anyways an obvious shortcoming is that it is very slow. Another solution to upgrade a machine is removing all packages and reinstalling everything fresh. In my experience this is vastly faster, but less automatic. A system to automate this second approach would be both easy to create and probably useful.

One may mention a port called portindex, which was introduced in 2004 by Radim Kolar and deleted soon after. It was an attempt to build a collection of python scripts allowing to reproduce part of the functionality of the make based system.

Finally, in order to evaluate the qualities and shortcomings of the FreeBSD ports system, it is useful to compare it with the system which benefits of the best appreciation in the Linux world, the Debian package system. The Debian project has the same age as the FreeBSD project, it has been founded in 1993 by Ian Murdock, followed by Bruce Perens, but first good releases date back to 1995-6. Equivalent tools to the FreeBSD pkg_* tools, dpkg and dselect (to be compared to the corresponding module of sysinstall) appeared in 1995.

Similarly as the FreeBSD system, the Debian system has source packages, which are not popular among end users, and binary precompiled packages, called package.deb which can be installed by running `dpkg -i package.deb`. Like the FreeBSD packages these packages contain besides the main content, some metatdata which is used by dpkg to build data structures on disk allowing to know the installed software and its network of dependencies.

But the main breakthrough came with the introduction of apt-get in 1998, which uses the information stored on disk, and the information stored in the Debian repositories to propose a list of packages to upgrade, and a correct dependency order to do it, in such a way as to be absolutely **reliable**. As long as one is willing to stay with a “stable” version of Debian (which frequently means obsoleted), keeping his machine current is no more complicated than running:
`apt-get update`
`apt-get upgrade` or `dist-upgrade`
The first command allows to discover progress in the package repositories, the second upgrades the machine.

Indeed it is the experience of many people that this works as intended and one can be confident that no bad surprise will occur. Of course as soon as one wanders in the realm of less well tested repositories, such as Debian Unstable, breakage can occur exactly as with portupgrade. A clear advantage of Debian is that the apt-get system (or its recent variant aptitude which is used in exactly the same way, aptitude update, etc.) has normal speed, and works predictably. When coupled with a more dynamic and adventurous management of software repositories, it has led to the considerable success of the Ubuntu distribution. We will try to discuss the main differences with the FreeBSD system and the lessons that can be learned to improve it.

One could also study another popular system, the Gentoo portage build system, which is said to be very much inspired by the FreeBSD system, but with the central role of the make tool replaced by python scripts. One can see the main “portage” python script here [4], which more or less reproduces the functionality of bsd.port.mk in python. It has 6965 lines of python against 5658 lines for bsd.port.mk, so it is quite similar in size.

An advantage is certainly that it is one program, and not something continually forking shell scripts, perl scripts, inferior makes, etc. so it should be faster. A bird's view of the script shows however that one has to reimplement a lot of things which are taken for granted with shell commands. But it is probably easier to understand, less hackish than the bsd variant. At the moment is developed a C++ version of part of the system, called "paludis". By total lack of knowledge of the Gentoo distribution i shall skip any discussion of its relative advantages.

2 The ports.

2.1 The ports hierarchy.

Almost all the intelligence defining the FreeBSD ports system is located under the port /usr/ports that we shall call portdir because it can be relocated. This intelligence is decentralized basically in two places: each port has a Makefile containing part of the game, but the essential complexity lives under /usr/ports/Mk. It is a collection of very intricate Makefiles which are included by each port Makefile and really allow the system to work. The main makefile is bsd.port.mk which contains almost all the procedures to build a package. If said package is for example a gnome package, it will include bsd.gnome.mk which contains further tools appropriate for gnome programs.

The ports themselves are organized into subdirectories that we shall not call categories because categories refer to a different concept in the system. Apparently, subdirectories and categories are the same thing in the NetBSD pkgsrc. At present these subdirectories are:

```
    accessibility arabic archivers astro audio benchmarks biology cad
chinese comms converters databases deskutils devel distfiles dns editors
emulators finance french ftp games german graphics hebrew hungarian irc
japanese java korean lang mail math mbone misc multimedia net net-im
net-mgmt net-p2p news packages palm polish portuguese print russian
science security shells sysutils textproc ukrainian vietnamese www x11
x11-clocks x11-fm x11-fonts x11-servers x11-themes x11-toolkits x11-wm
```

but new subdirectories are created and perhaps other disappear. Under each one, one finds a Makefile, which allows to build all corresponding packages, and the ports proper. For example, under `accessibility`, which is small, one has:

```
    Makefile at-spi atk dasher gail gnomemag gnomespeech gnopernicus gok
kdeaccessibility linux-atk ruby-atk
```

Many subdirs have of the order of 500 ports, for a grand total of more than 15000 ports at present, to be compared to the 17000 packages of Debian. In other words, FreeBSD and Debian are the most comprehensive systems of free software distribution.

2.2 Anatomy of a port.

We are going to explain how a port looks like by taking the example of the atk port in the above directory "accessibility". First atk is a directory and contains the following general structure:

```
Makefile distinfo files/ pkg-descr pkg-plist
```

The main element is of course the Makefile, distinfo contains MD5 and SHA256 hashes of the corresponding source files, allowing to identify them, pkg-descr is a description of the program, ending if possible by a line identifying the main web site of the program. For example here it is:

```
     The GNOME Accessibility Toolkit (ATK) contains accessibility widgets,
just like GTK contains GUI element widgets.  However, ATK's widgets are
platform-independent, so they can be used with Qt, Motif, etc.

WWW: http://developer.gnome.org/projects/gap/
```

Then we have pkg-plist, which gives the so–called "packing list" of the port, that is essentially the list of all the files the port installs, plus some administrative data pertaining to installation and desisntallation. Writing the packing list is a tedious and error prone task for the port maintainer. In this domain Debian offers much more automated tools, hence reducing the risk of an error.

Finally we have the directory files in which can be found various files such as patches allowing the program to build. In the present case there are no patches to the source code, but a patch to the Makefile.in, a component of the "configure" script for the software.

2.3 The make tool.

The ports system is entirely based on the BSD variant of the make program. Let us recall for the benefit of the non programmer that generically a make program is a sort of automaton which works by decomposing a goal into subgoals, called "targets" and giving rules to fulfill a target. So it is a rules based system. A typical makefile correspondingly looks like

```
target1: target2 target3
        <some shell commands>
target2: target4
        <other shell commands>
...
```

Make has the notion of temporal dependency order. If either target2 or target3 have been more recently executed than target1, then running make will execute target1. Make also uses variables which can be susbstituted elsewhere in the makefile. For example one can see the following:

```
CFLAGS="-g"
...
prog : prog.c
       cc ${CFLAGS} -o prog prog.c
```

The shell command, after substitution becomes `cc -g -o prog prog.c`. This corresponds to the most basic usage of make. However the BSD variant has much more than that (the GNU variant also has such extensions) it has conditionals of all sorts which allow to turn make into a complete programming language, and string substitution operators which allow to massage the variables to almost any degree possible. A complete but terse description of all those features can be found in the make man page, showing that great complexity can be hidden in only a few pages!

A very important point: when running "make", the system file `sys.mk` in the standard make directory `/usr/share/mk` is included defining a set of implicit rules which describe how to fulfill a target automatically, for example, how to obtain program.o from program.c by using the compiler. Hence simply stating that a target depends on object files is sufficient to invoke automatic compilation of source files. More to our point this file invokes loading `/etc/make.conf` in which one can put arbitrary makefile stuff, and in the simplest case define variables.

For example one can set here WITHOUT_GNOME=yes which will have effect on the building of ports which would have included optional Gnome support otherwise. A good example is provided by the gimp port in `/usr/ports/graphics/gimp` whose Makefile is carefully crafted to include gnome support if "gnome-panel" is reported to be present on the machine (this is the component of gnome which displays taskbars). In this case, without any intervention of the user, without having filled any configuration information, this bit of make sorcery includes gnome support in gimp, and as a result, gimp acquires dependency on gnome and the package produced gets a gnome suffix.

The only way to counter this tendency of maximal crap pollution is to set the WITHOUT_GNOME=yes variable in /etc/make.conf or run make while defining this variable in any way, as environment variable works. Clearly this has big importance for upgrading, notably because gimp-gnome packages don't exist in the package repositories and is totally hermetic to newcomers to FreeBSD if not experienced users.

```
niobe% make -V PKGNAME
gimp-gnome-2.2.10_1,1
niobe% make -V PKGNAME WITHOUT_GNOME=yes
gimp-2.2.10_1,1
```

The problem is made worse since there is no explicit list of variables or targets that the user may tweak. Some of them are listed in a long comment at the beginning of bsd.port.mk, but the list is not complete. This is a point where the ports system is obfuscated, perhaps intentionally.

All this magic is used in the main driving makefile of the ports system, `bsd.port.mk`. For example, to build a package it is necessary to do in correct order many steps, such as fetching source code, checking its MD5 sum, extract it, patch the source code, run the configure script if present, build the package, etc. The order is maintained thanks to the above temporal dependency of targets. For example we find:

```
# Disable checksum
.if defined(NO_CHECKSUM) && !target(checksum)
checksum: fetch
        @${DO_NADA}
.endif

# Disable build
.if defined(NO_BUILD) && !target(build)
build: configure
        @${TOUCH} ${TOUCH_FLAGS} ${BUILD_COOKIE}
.endif
```

This allows to define variables which inhibit parts of the build and enforce dependencies. The idiom `.if !target(checksum)` means, if the target "checksum" is not defined then define it as in the line below. In this line it depends on fetch, so before computing the checksum one has to fetch the source code. In this case if the variable NO_CHECKSUM has been defined then one does nothing more to fulfill the target. If on the other hand the target checksum has been defined previously, perhaps in the ports makefile, then nothing here is executed. This allows to special case things.

To give a final example of the power of the conditional constructs occurring in make, and its subshells, here is the way patches are performed:

```
.if defined(PATCHFILES)
     @${ECHO_MSG} "===>  Applying distribution patches for ${PKGNAME}"
     @(cd ${_DISTDIR}; \
       for i in ${_PATCHFILES}; do \
           if [ ${PATCH_DEBUG_TMP} = yes ]; then \
                   ${ECHO_MSG} "===>   Applying distribution patch $$i" ; \
           fi; \
           case $$i in \
                   *.Z|*.gz) \
                           ${GZCAT} $$i | ${PATCH} ${PATCH_DIST_ARGS}; \
                           ;; \
                   *.bz2) \
                           ${BZCAT} $$i | ${PATCH} ${PATCH_DIST_ARGS}; \
                           ;; \
                   *) \
                           ${PATCH} ${PATCH_DIST_ARGS} < $$i; \
                           ;; \
           esac; \
       done)
.endif
```

Conditionals in the shell command allow to chose the correct unzipping program for compressed patches, either gzcat or bzcat in case gzip or bzip2 has been used, before applying patch. All newlines are escaped so that the shell command looks like one line to the make program. This also serves to illustrate the considerable number of subshells and other programs which make spawns when running, which leads us to one problem of the FreeBSD system, this make mechanism is slow. Simply running make without any compilation in a port directory easily takes from 0.1 to 1s. according to the complexity of the port and the speed of the computer. When multiplied by the 15000 ports this may mean something of the order of an hour or more.

We can now come back to our subject, explaining the Makefile in a port, and we specifically illustrate that on the Makefile of atk.

```
PORTNAME=       atk
PORTVERSION=    1.10.3
PORTREVISION=   1
CATEGORIES=     accessibility devel
MASTER_SITES=   ${MASTER_SITE_GNOME} \
                ftp://ftp.gtk.org/pub/gtk/v2.6/
MASTER_SITE_SUBDIR=     sources/${PORTNAME}/1.10
DIST_SUBDIR=    gnome2

MAINTAINER=     gnome@FreeBSD.org
```

```
COMMENT=        A GNOME accessibility toolkit (ATK)

USE_BZIP2=      yes
INSTALLS_SHLIB= yes
USE_AUTOTOOLS=  libtool:15
USE_GNOME=      gnomehack glib20 ltverhack
CONFIGURE_ARGS= --enable-static \
                --disable-gtk-doc \
                --with-html-dir=${PREFIX}/share/doc
CONFIGURE_ENV=  CPPFLAGS="-I${LOCALBASE}/include" \
                LDFLAGS="-L${LOCALBASE}/lib"

post-patch:
        @${REINPLACE_CMD} -e "/^SUBDIRS =/s/tests//" \
                ${WRKSRC}/Makefile.in

.include <bsd.port.mk>
```

We can see that the Makefile mainly consists of the definition of variables, there is just one target which allows to special case post-patch. But the main magic here is the last line, which includes bsd.port.mk, and with that the whole bag of tricks that it contains. However since variables have been defined, the rules in bsd.port.mk will see these variables (because the inclusion is at the end) and react accordingly. In particular since the variable USE_GNOME is set, this will include almost at the beginning the file bsd.gnome.mk which will considerably modify the subsequent logic in bsd.port.mk. Remark that the logic of these makefiles is always backwards.

As far as we are more interested in the logic of the dependencies between the ports and packages than the actual building of the package, and since the porter's handbook [5] has complete information on this part of the subject, we shall just stress the importance of a certain number of variables occurring here. We have obviously PORTNAME which gives the name of the port, PORTVERSION which gives the version number as given by the upstream developer of the software, PORTREVISION which is the revision number of the FreeBSD port itself, for example a new patch has been introduced, we may have PORTEPOCH in case PORTVERSION is flawed, we may also have prefixes and suffixes, which may be introduced automatically by system makefiles. For example bsd.gnome.mk may introduce a gnome suffix, and bsd.python.mk may introduce a py24 prefix (which would have been py23 with an earlier revision of the system) such as in py24-tkinter-2.4.2_1 when the portname was simply tkinter, and the port path was x11-toolkits/py-tkinter.

All this boils down to the fact that when compiling the port we end up with a package which has a name, called PKGNAME differing considerably from PORTNAME. This is one of the main facts in the port-package interaction, the package name is built automatically, through intricate interactions in the system makefiles, and it is impossible to discover it in a reliable way without running the make program. Running make it is very easy to obtain it however this way:

```
rose% cd /usr/ports/accessibility/atk
rose% time make -V PKGNAME
atk-1.8.0
make -V PKGNAME  0,38s user 0,09s system 32% cpu 1,435 total
```

We see two things here. First, on this machine, the name of the package will be atk-1.8.0. Second, this machine is a laptop equipped with a Celeron 400, and this simple command takes 1.5s of clock time to run! To build the complete mapping between port names and package names on such a machine will take several hours. The rule for the formation of the name is that the port name is eventually prefixed and suffixed, such as in "emacs-nox" or "gimp-gnome" or "py24-tkinter", then is added PORTVERSION separated by hyphen, then PORTREVISION separated by underscore if it does not vanish (default value is 0) and finally PORTEPOCH separated by comma if it doesn't vanish (default value 0). We finally get package names such as vorbis-tools-1.0.1_3,3. Incidentally remark that such a name like vorbis-tool containing hyphen is difficult to parse, with respect to prefixes and suffixes.

Besides the naming of the package, the Makefile also introduce either explicitly or implicitly (for example through inclusion of bsd.gnome.mk) dependencies between ports. For example a port may need a library to provide functionality, this is called LIB_DEPENDS. A port may need GNU make to build, which is called gmake in FreeBSD. Hence we have gmake appearing in BUILD_DEPENDS. Similarly we have RUN_DEPENDS, for example mplayer may need some extra codecs to run which may be provided in another port. These are the 3 main types of dependencies, there are 3 other ones, FETCH_DEPENDS, EXTRACT_DEPENDS, and PATCH_DEPENDS whose meaning is obvious. We can discover all of them by running make -V as above, moreover we can get them all simultaneously by putting several -V options on the command line, which is not much more expensive that a single invocation. Now that the system cache has been primed i get for the same port atk:

```
rose% time make -V RUN_DEPENDS -V LIB_DEPENDS -V BUILD_DEPENDS
pkg-config:/usr/ports/devel/pkgconfig                   # RUN
glib-2.0.400:/usr/ports/devel/glib20                    # LIB
/usr/local/bin/libtool15:/usr/ports/devel/libtool15     # BUILD
                              pkg-config:/usr/ports/devel/pkgconfig
make -V RUN_DEPENDS -V LIB_DEPENDS  0,36s user 0,09s system 95% cpu
0,474 total
```

The library dependency shows that it is a dependency on the port devel/glib20, but furthermore on the version 2.0.400 of the library. The run time dependency is only on pkg-config without further specification. Note that these dependencies have been obtained implicitly through the use of the USE_GNOME clause.

A summary of these informations may be obtained by using the describe target, this time on a recent machine (P4 3Ghz):

```
rose% make describe
atk-1.10.3_1|/usr/ports/accessibility/atk|/usr/local|A GNOME
accessibility toolkit (ATK)|
/usr/ports/accessibility/atk/pkg-descr|gnome@FreeBSD.org|
accessibility devel||/usr/ports/devel/libtool15||
/usr/ports/devel/glib20 /usr/ports/devel/libtool15
/usr/ports/devel/pkgconfig|
/usr/ports/devel/glib20 /usr/ports/devel/pkgconfig|
http://developer.gnome.org/projects/gap/
make describe  0,09s user 0,05s system 96% cpu 0,142 total
```

This is in fact one line with 14 fields separated by the pipe symbol — and which are, in order:

1. “pkgname”
2. complete port “path”
3. “prefix” where installation is performed
4. “comment” in short form
5. “description” file
6. email address of the “maintainer”
7. “categories” to which the port belongs
8. extract dependencies
9. patch dependencies
10. fetch dependencies
11. build dependencies
12. run dependencies
13. lib dependencies
14. “website” of the software

In the example of atk, there is no fetch or extract dependency but libtool is registered as patch dependency as well as build dependency. In fact examining the target "describe" in bsd.port.mk, which is implemented by an inlined perl script, one discovers that the library dependency is added to both the run and build dependencies. It is easy to see that in the present example, glib20 has been added to the run dependency and the build dependency. It is interesting to look at the code snippet doing that:

```
.if !target(describe)
describe:

    ......

    @perl -e ' \
        if ( -f q{${DESCR}} ) { \
            print q{|${DESCR}}; \
        } else { \
            print q{|/dev/null}; \
        } \
        print q{|${MAINTAINER}|${CATEGORIES}|}; \
        @edirs = map((split /:/)[1], split(q{ },q{${EXTRACT_DEPENDS}})); \
               # picks the second part, the path, after : in a dependency
        ......

        for (@bdirs, @ddirs, @ldirs) { \
             $$xb{$$_} = 1; \
        } \                    # Joins build_deps, deps (obsoleted),lib_deps
        print join(q{ }, sort keys %xf), q{|}; \

        .......

        print qq{\n};'
.endif
```

The dependencies here are specified only in terms of port's path, which is essentially the same as the PORTNAME, and not in terms of package names. These informations can be obtained locally in the considered port without knowing detailed things about the other ports. Note that the COMMENT is defined in the Makefile as well as the CATEGORY. Also that atk belongs to two categories, accessibility, and devel, who happen to be subdirs of /usr/ports, but there are other categories. The complete list of "valid categories" at present is:

```
     accessibility, afterstep, arabic, archivers, astro, audio,
benchmarks, biology, cad, chinese, comms, converters, databases,
deskutils, devel, dns, editors, elisp, emulators, finance, french, ftp,
games, german, gnome, graphics, haskell, hebrew, hungarian, ipv6, irc,
japanese, java, kde, korean, lang, linux, lisp, mail, math, mbone, misc,
multimedia, net, net-mgmt, news, offix, palm, parallel, pear, perl5,
picobsd, plan9, polish, portuguese, print, python, ruby, russian, scheme,
science, security, shells, sysutils, tcl80, tcl81, tcl82, tcl83, tcl84,
textproc, tk80, tk82, tk83, tk84, tkstep80, ukrainian, vietnamese,
windowmaker, www, x11, x11-clocks, x11-fm, x11-fonts, x11-servers,
x11-themes, x11-toolkits, x11-wm, xfce, zope
```

Categories are used in particular to help choose packages in FreeBSD software repositories. Sysinstall shows them when installing from cdrom or ftp, etc.

Finally there is another way to discover dependencies which leads to very different result. It is using the pretty-print targets. Still in the atk directory we obtain:

```
niobe% make pretty-print-run-depends-list
This port requires package(s) "gettext-0.14.5_2 glib-2.8.6_1
libiconv-1.9.2_2 libtool-1.5.22_2 perl-5.8.8 pkgconfig-0.20" to run.
niobe% make pretty-print-build-depends-list
This port requires package(s) "gettext-0.14.5_2 glib-2.8.6_1
libiconv-1.9.2_2 libtool-1.5.22_2 perl-5.8.8 pkgconfig-0.20" to build.
```

The first obvious difference is that now dependencies are labelled by package names, and not by port paths. In particular they contain version numbers of the software. This can be used to know on which particular version of some software another depends, and is necessary to know when upgrading. Second, we see in these lists packages which were not present in make describe, such as libiconv-1.9.2_2 or perl-5.8.8. These packages have been introduced by **recursive dependency** analysis, that is they are dependencies of dependencies and as such necessary to run or build the considered package. More precisely it is **run dependencies** of tools (whether it is extract, fetch, patch or build) which are recursively expanded, because if a port needs gmake to build it does not in any way require build tools or fetch tools of gmake to build, but it requires all run tools dependencies that gmake needs. To perform the mapping of ports to package names or recursive expansion of run dependencies requires non local knowledge of the ports, in fact potentially knowledge obtained by running make describe on all the ports of the system. Clearly this knowledge requires very long time to gain, and is cached in an indexfile /usr/ports/INDEX, which contains, for each port a line similar to the above make describe, but with 13 fields, in a slightly different order, and dependencies in the form presented by the above pretty-print commands. In fact, pretty-print extracts its information from INDEX.

As one may easily note from the previous discussion, obtaining this INDEX file in an efficient way is a central problem of the system. Since it is a cache it has the same problems as many other caches, namely it may be inconsistent with the actual state of the ports system, which yields incoherence when using dependency lists. More on this later on.

2.4 Building a package.

Typing “make” in the atk directory triggers things such as the following:

```
=> atk-1.10.3.tar.bz2 doesn’t seem to exist in /usr/ports/distfiles/gnome2.
=> Attempting to fetch from
ftp://ftp.belnet.be/mirror/ftp.gnome.org/sources/atk/1.10/.
atk-1.10.3.tar.bz2                              100% of  529 kB 1844 kBps
===>  Extracting for atk-1.10.3_1
=> MD5 Checksum OK for gnome2/atk-1.10.3.tar.bz2.
=> SHA256 Checksum OK for gnome2/atk-1.10.3.tar.bz2.
===>  Patching for atk-1.10.3_1
===>   atk-1.10.3_1 depends on file: /usr/local/bin/libtool - found
===>  Applying FreeBSD patches for atk-1.10.3_1
===>   atk-1.10.3_1 depends on file: /usr/local/bin/libtool - found
===>   atk-1.10.3_1 depends on executable: pkg-config - found
===>   atk-1.10.3_1 depends on shared library: glib-2.0.0 - found
===>  Configuring for atk-1.10.3_1
checking for a BSD-compatible install... /usr/bin/install -c -o root -g wheel
checking whether build environment is sane... yes
checking for gawk... gawk
.......
===>  Building for atk-1.10.3_1
make  all-recursive
Making all in atk
.......

creating libatk-1.0.la
(cd .libs && rm -f libatk-1.0.la && ln -s ../libatk-1.0.la libatk-1.0.la)
Making all in docs
Making all in po
```

We see that the successive stages of the process are “fetching”, “extracting”, “patching”, “configuring”, “building”, and of course at the end “installing”.

The source code is first downloaded in `gnome2/atk-1.10.3.tar.bz2` under the subdirectory `distfiles` of portdir. The system uses the information obtained from MASTER_SITES to locate a server offering the source, then from MASTER_SITE_SUBDIR it knows where to find it, and finally it uses programs such as “fetch”, “ftp”, “wget” or "curl" to download it, according to fetch dependencies. Each stage can be run independently, one can type `make fetch` to only fetch source code. Of course dependencies are obeyed, so that `make extract` will first run make fetch. Here MASTER_SITE_SUBDIR is sources/atk/1.10/ and the system uses MASTER_SITE_GNOME to locate a gnome mirror.

```
niobe% make -V MASTER_SITE_GNOME
ftp://ftp.belnet.be/mirror/ftp.gnome.org/%SUBDIR%/ ....
```

This gives a long list of mirrors which are tried in turn. Of course SUBDIR is replaced by "sources/atk/1.10/" leading to the URL appearing above. The program fetch is then invoked by default, it accepts URLs of the form ftp:// or http:// and will retreive the source code in distfiles/gnome2 because DIST_SUBDIR=gnome2 is specified. The name of the source code file is not specified in the Makefile, it is computed by the system and is found in the variable DISTFILES. It derives from DISTNAME which is atk-1.10 not surprisingly from already known data (port name and port version) and EXTRACT_SUFX which turns out to be .tar.bz2 as a default value when USE_BZIP2 is set, which is the case in the Makefile.

Note one can set the DISTDIR variable to some user writable file instead of distfile to allow building in user mode, which also needs to set WRKDIR to a user writable place (this is the usual subdirectory work in a port):

```
niobe% make DISTDIR=/tmp/d  WRKDIR=/tmp/w
=> atk-1.10.3.tar.bz2 doesn't seem to exist in /tmp/d/gnome2.
...
creating libatk-1.0.la
(cd .libs && rm -f libatk-1.0.la && ln -s ../libatk-1.0.la libatk-1.0.la)
Making all in docs
Making all in po
```

If dependencies had been required they would have been built previously. This is one of the main strengths of the ports system, **dependencies** are resolved automatically when building a port. Note that libtool is indeed a patching dependency and it is checked at the patching stage.

Similarly the MD5 and SHA256 hashes are checked before extracting the tarball for atk, this is the checksum target on which the extract target depends. One can disable this "security" feature by setting the variable NO_CHECKSUM, so allowing to build a different version of the source code. By the way on can enable a check of port vulnerabilities in the port tree, using an external base of known vulnerabilities. This is obtained by installing the "portaudit" port, which uses the database `auditfile.tbz` in `/var/db/portaudit`. It may be convenient to disable this check for building a particular vulnerable software by setting the variable DISABLE_VULNERABILITIES.

After extraction the files are patched (one can type `make patch` to get to this stage) and the port is then configured, `make configure` which in many cases means that the "configure" script is run. In some cases OPTIONS are defined in the Makefile, which prompts the user for a choice of such options, using a special dialog. These options will then be fed to the configure script or to some other script. The net effect is to set WITH_something or WITHOUT_something variables which are used in further processing stages. An example of a port which has a multitude of such options is "mplayer-skins". In its Makefile one finds:

```
OPTIONS=        SKIN_ALL "all skins" off
OPTIONS+=       SKIN_DEFAULT "the default MPlayer skin" on
....
OPTIONS+=       SKIN_XMMPLAYER "XMMS lookalike" off

.include <bsd.port.pre.mk>
```

This is the required syntax for options, including the call to bsd.port.pre.mk. The choices are then preserved in `/var/db/ports` so that subsequent builds are no more interactive. The targets "rmconfig" and "showconfig" allow to tweak these settings. One can also set the variable BATCH to not be prompted, a thing that some people find irritating. Then default values are silently accepted.

Another possibility is to fill all options forms in advance for the port to be compiled and its dependencies. For doing that, `make config-recursive` will allow to do an unattended build afterwards.

The "build" stage is then processed, `make build`, and should normally end with the software fully compiled. At this stage the atk directory contains a "work" subdirectory in which everything has been built:

```
niobe# ls -a work
.                                        atk-1.10.3
..                                       gnome-libtool
.build_done.atk-1.10.3_1._usr_local      gnome-libtool.bak
.configure_done.atk-1.10.3_1._usr_local gnome-ltmain.sh
.extract_done.atk-1.10.3_1._usr_local    gnome-ltmain.sh.bak
.patch_done.atk-1.10.3_1._usr_local
```

In particular we see here files such as ".extract_done" which are used to timestamp the successive targets, and the directory "atk-1.10.3" in which the build really occurs. In this particular case there are also libtool files. If we type make extract, the command will return immediately since it will discover the presence of .extract_done indicating the target is fulfilled. Erasing this file allows to make extraction again.

Now the “atk” package can be installed by running:

```
make install
```

Before installing a package the system checks for **conflicts**, that is checks if it will not overwrite an older version. This may cause considerable trouble, because if we deinstall the old version, we may remove shared libraries which may be required by some of our installed packages and so break them. The target “deinstall” is used to remove the old version. One of the selling points of portupgrade is that it will preserve old shared libraries. I have suffered a lot of grief from this “feature” that i consider to be buggy. Anyways studying bsd.port.mk reveals an antidote, one has to set the variable DISABLE_CONFLICTS. Then the new port will silently overwrite the old one, which will preserve the old shared libraries having a different version number, but will install new executables, and corresponding documentation. The port or package is also inspected for suid files and similar hazards and a security report is emitted on console.

The installation target may also trigger some install time configuration, such as installing a new user in the passwd file, initializing some data bases, etc. This is achieved by defining a post-install target in the Makefile which does the required job, or writing a pkg_install file in the port directory, etc. The procedure is clearly less formalized than in the Debian case who has a systematic procedure of “package configuration” after each package installation, which can be accessed through dpkg-preconfigure, dpkg-reconfigure, debconf. These procedures have a good reputation of reliability, the FreeBSD solutions are more hackish and error prone.

If keeping a copy of the package is desired one can run:

```
make package
```

which deposits a copy of the compressed package locally, or in the subdir `packages/All` of portdir if it exists. It will be named “atk-1.10.3_1.tgz” or “atk-1.10.3_1.tbz” according to the compression program used, gzip or bzip2. In the next section we describe the content of this package.

Finally we run:

```
make clean
```

to clean the port directory, which removes recursively the work subdirectory. Moreover it cleans all build dependencies, as we can see:

```
niobe# make clean
===>  Cleaning for libtool-1.5.22_2
===>  Cleaning for pkgconfig-0.20
===>  Cleaning for glib-2.8.6_1
===>  Cleaning for gmake-3.80_2
===>  Cleaning for perl-5.8.8
```

```
===>  Cleaning for gettext-0.14.5_2
===>  Cleaning for libiconv-1.9.2_2
===>  Cleaning for atk-1.10.3_1
```

Notice two things: first some stages of the process, like building the package dependency list or cleaning dependencies require recursive dependency expansion. The makefile bsd.port.mk has provisions to do such things locally via embedded shell scripting. Here for cleaning, the dependencies are generated by "make all-depends-list". To build this list one first collect all direct dependencies (fetch, patch, extract, lib, run, build) in a variable _UNIFIED_DEPENDS, and then the path part is extracted into _DEPEND_DIRS. Then one goes to each of these paths and similarly obtain direct dependencies. This recursion is typically obtained this way:

```
ALL-DEPENDS-LIST= \
    L="${_DEPEND_DIRS}";\
    while [ -n "$$L" ]; do \
    .....
       for d in $$L; do \
    .....
          if ! children=$$(cd $$d && ${MAKE} -V _DEPEND_DIRS); then\
...
```

This calls make -V recursively, indefinitely, except that the recursion stops when the variable becomes empty. Note that all dependencies are expanded, not only run time dependencies. Hence one is sure to clean everything. This is necessary because potentially all dependencies including the build, etc. dependencies, have been compiled recursively and have a work directory to be cleaned.

A second thing to note is that we need to run that as root, if only to write the source package in the distfile directory, and also in some cases to fix permissions and ownerships in some files of the package. NetBSD pkgsrc and Debian have provisions to completely build packages as ordinary users, Debian uses a command "fakeroot" to do that.

3 The packages.

We now examine how the FreeBSD system uses the packages we have just produced. A package is simply the compressed tar archive of the compiled software to be installed, plus some files containing metadata. For example we have:

```
niobe% tar tvfz atk-1.10.3_1.tgz|more
```

```
-rw-r--r--  0 root   wheel   10504  9 mai 23:18 +CONTENTS
-rw-r--r--  0 root   wheel      36 16 mar 03:16 +COMMENT
-rw-r--r--  0 root   wheel     252 16 mar 03:16 +DESC
-r--r--r--  0 root   wheel   15305 16 mar 03:16 +MTREE_DIRS
-r--r--r--  0 root   wheel    1446 16 mar 03:16
 include/atk-1.0/atk/atk-enum-types.h
-r--r--r--  0 root   wheel    1603 16 mar 03:16 include/atk-1.0/atk/atk.h
 ....
```

The metadata files are the first four files, prepended by +. Upon installation the metadata are installed in the package database, while the other files, properly pertaining to the software are installed under prefix, usually under /usr/local.

3.1 The package database.

The package database is comprised of directories, one for each package, containing the above metadata, and located under `/var/db/pkg` usually. So listing this directory one gets immediately all packages installed in the system. Looking inside a specified port one gets its metadata. For example:

```
niobe% ls /var/db/pkg/atk-1.10.3_1
+COMMENT      +CONTENTS     +DESC          +MTREE_DIRS   +REQUIRED_BY
```

Note first that the package directory is named after the package name and not the port name. So the port tree displays the name of ports, while the package database displays name of packages. Note also there is a new file appearing `+REQUIRED_BY`. The others are the same files as in the packages. Here COMMENT is simply the short comment appearing in the Makefile and DESC is the description file in the ports, MTREE_DIRS is specific data which FreeBSD uses to fix permissions using the mtree program (see man mtree). The most interesting file is +CONTENTS which derives more or less from the packing list in the port, but augmented with a lot of infos by the package building procedure.

```
niobe% more ./+CONTENTS
@comment PKG_FORMAT_REVISION:1.1
@name atk-1.10.3_1
@comment ORIGIN:accessibility/atk
@cwd /usr/local
@pkgdep pkgconfig-0.20
@comment DEPORIGIN:devel/pkgconfig
```

```
@pkgdep perl-5.8.8
@comment DEPORIGIN:lang/perl5.8
@pkgdep libiconv-1.9.2_2
@comment DEPORIGIN:converters/libiconv
@pkgdep gettext-0.14.5_2
@comment DEPORIGIN:devel/gettext
@pkgdep glib-2.8.6_1
@comment DEPORIGIN:devel/glib20
include/atk-1.0/atk/atk-enum-types.h
@comment MD5:0f772e1c46210a0e97a019ad0d2472dc
....
```

All the files comprising the software are checksummed as a measure of caution. It allows in particular to prevent removal of a file which has been user modified. The important header of the file presents the package name, here the port "accessibility/atk" which allowed to build the package, and is called the **origin** of the package, then the installation prefix, /usr/local, followed by run time dependencies which are remarkably provided each with its origin. We have explained in the previous section how such information can be gained in the package building step. It is exactly the output of the target "package-depends-list" . This dependency list is computed basically in the same way as the preceding one, except that only direct run time dependencies are considered, that is the union of LIB_DEPENDS and RUN_DEPENDS. In our case since libtool doesn't appear in either of these lists, it doesn't appear in the final result. This is different of the output of "make pretty-print-run-depends-list". In this case the output comes from the INDEX file, which in turns comes from "make describe" and recursive expansion. Let's do it by hand:

```
niobe% pwd
/usr/ports/accessibility/atk
niobe% make pretty-print-run-depends-list
This port requires package(s) "gettext-0.14.5_2 glib-2.8.6_1
libiconv-1.9.2_2 libtool-1.5.22_2 perl-5.8.8 pkgconfig-0.20" to run.
niobe% make describe|awk -F\| {'print $12'}        # The RUN field.
/usr/ports/devel/glib20 /usr/ports/devel/pkgconfig
niobe% (cd /usr/ports/devel/glib20;make describe)|awk -F\| {'print $12'}
/usr/ports/devel/gettext /usr/ports/devel/pkgconfig
/usr/ports/lang/perl5.8
niobe% (cd /usr/ports/devel/gettext;make describe)|awk -F\| {'print $12'}
/usr/ports/converters/libiconv
niobe% (cd /usr/ports/converters/libiconv;make describe)|
                                          awk -F\| {'print $12'}

niobe% (cd /usr/ports/devel/pkgconfig;make describe)|
                                          awk -F\| {'print $12'}
```

```
niobe% (cd /usr/ports/lang/perl5.8;make describe)|awk -F\| {'print $12'}
```

at which point recursion stops and libtool doesn't appear in the run times dependencies, which is normal because it is a build time dependency. In fact the INDEX is correct, but the target "pretty-print-run-depends-list" is bogus, it prints the same eighth field of INDEX as the target "pretty-print-build-depends-list" while it should print ninth field. Note, this is true for FreeBSD-6.1-RELEASE but is corrected in revision 1.533 of bsd.port.mk dated Tue May 23 21:53:18 2006.

We now turn to the last file +REQUIRED_BY which doesn't list the dependencies of the current package (the "downwards" dependencies) but all the installed packages which require the current package as a dependency (the "upwards" dependencies). Obviously this information is entirely dependent on the actually installed packages and is filled only when installing some new package depending on this one. This occurs as part of the procedure called **registring installation** when a package or a port is installed. To understand how it works, consider the target "fake-pkg" in bsd.port.mk.

```
.if !target(fake-pkg)
fake-pkg:
  ....
  ${ECHO_MSG} "===>   Registering installation for ${PKGNAME}"; \
  ....
  for dep in `${PKG_INFO} -qf ${PKGNAME} | ${GREP} -w ^@pkgdep | ${AWK}\
                      '{print $$2}' | ${SORT} -u`; do \
      if [ -d ${PKG_DBDIR}/$$dep -a -z `${ECHO_CMD} $$dep | ${GREP} -E
                                     ${PKG_IGNORE_DEPENDS}` ]; then \
          if ! ${GREP} ^${PKGNAME}$$ ${PKG_DBDIR}/$$dep/+REQUIRED_BY \
               >/dev/null 2>&1; then \
               ${ECHO_CMD} ${PKGNAME} >> ${PKG_DBDIR}/$$dep/+REQUIRED_BY;\
          fi; \
      fi; \
  done; \
...
.endif
```

In this shell script snippet `pkg_info -qf pkgname` in fact produces the package plist file, which when grepping gives the following, (where awk selects the second field, that is a package name):

```
niobe%  pkg_info -qf -x atk-1.10|grep -w ^@pkgdep
@pkgdep pkgconfig-0.20
```

```
@pkgdep perl-5.8.8
@pkgdep libiconv-1.9.2_2
@pkgdep gettext-0.14.5_2
@pkgdep glib-2.8.6_1
```

the run time dependencies of atk, as listed by the target package-depends-list. Looking for each one under the corresponding directory of PKG_DBDIR (that is /var/db/pkg), for example in the subdir perl-5.8.8, and if not present we add our current package, here atk, to the list +REQUIRED_BY of upwards dependencies. In other words we specify in the perl directory that atk requires perl, which means we perform an “inversion” of the dependency order relation. This inversion is only performed on installed packages, however.

3.2 The package tools.

The package tools are the last part of the system originally conceived by J. Hubbard and whose power and intricacy is absolutely remarkable. They are comprised of a number of utilities, written in C, allowing to manage easily various tasks belonging to package management. In general they deal with objects and data located under /var/db/pkg, and have no interaction with things under /usr/ports. The source code for these utilities is under `usr.sbin/pkg_install` in the source directory `/usr/src`. They are pkg_add, pkg_create, pkg_delete, pkg_version, pkg_info. In the same way that FreeBSD packages are not very different from Linux rpm or deb packages, these tools are analogous to the corresponding Linux tools. For example pkg_install does the same thing as dpkg -i for Debian or rpm -i for Redhat, it installs a package and at the same time updates the metadata on the machine saying that the package is installed. Of course pkg_add has further functionality such as automatic installation of dependencies that the Linux tools don't have.

A nice goodie is that one can install packages from FreeBSD package repositories in a very concise way. Suppose i want to install firefox, i don't know the version number, and i have internet connexion. I have only to type:
`pkg_add -r firefox`
then the program will figure out from “uname” what version of the OS i am using, and will ask the main FreeBSD ftp server what package for firefox is in stock for this OS version, will download and install it. Recursively it will do the same for all dependencies of firefox. If it cannot find a precompiled appropriate package, it will return doing nothing. One can set environment variables to query other ftp or http servers.

Of course package_delete does the inverse, it deletes packages, such as the Redhat rpm -e or the Debian dpkg -r. It has some notable features, such as providing the name of the package to be deleted as a regular expression, which allows to ignore version numbers, but is also a bit dangerous. It features a recursive removal option, package_delete -r which deletes packages which depend on the given package. Since we remove it, they will be broken anyways, so this makes sense. These are the packages in +REQUIRED_BY.

As with similar options of rpm and dpkg, pkg_create allows to build packages. In particular there is the very convenient option -b which allows ordinary users to create a package from an installed port:

```
niobe% pkg_create -x  -b atk
atk-1.10.3_1.tgz linux-atk-1.8.0_1.tgz
```

Of course this works by using the packing list in the database to know what files to feed to tar. This, in passing, illustrates the dangers of using globs. The man page is very informative about the format of the packing list file. Note that this tool is used, with the -O option, allowing to build packing file, by bsd.port.mk in the target "fake-pkg" quoted above, in the line, where PKG_CMD is in fact pkg_create, and we see that this invocation creates the +CONTENTS file.
`PKG_CMD PKG_ARGS -O PKGFILE > PKG_DBDIR/PKGNAME/+CONTENTS;`

The tool pkg_version is a more recent tool whose aim is to help in upgrading the installed packages. It lists the packages which need upgrading against the port tree. Typically for each package in the database of installed packages, it picks its origin, then goes to the corresponding port of the port tree (if it still exists - it may have disappeared or been moved) and runs make -V PKGNAME to get the package name of the possible upgrade. It is then easy to list the upgradable packages. The output is like that:

```
niobe%  pkg_version -v
ImageMagick-6.2.5.5_3              =   up-to-date with port
....
fr-openoffice.org-2.0.0            <   needs updating (port has 2.0.2.rc2)
freelibiberty-0.2_1                =   up-to-date with port
....
```

This is a primitive management system certainly superseded by portupgrade. It is also probably the only one which deals with /usr/ports, and not exclusively the package database.

The most interesting tool for our purposes is pkg_info. It allows to discover all values stored in the package database, in the same vein as rpm -q for RedHat or various options of dpkg, such as dpkg -I or dpkg -L for Debian. First the simplest, which lists all installed packages with short comments like dpkg -l.

```
niobe% pkg_info
ImageMagick-6.2.5.5_3 Image processing tools
ORBit2-2.12.5_2     High-performance CORBA ORB with support for the C language
....
```

With the -a option one gets a more detailed output, with short comment and description, such as the following, but for all installed packages.

```
niobe% pkg_info  R-2.2.1
Information for R-2.2.1:

Comment:
A language for statistical computing and graphics

Description:
....
R is a system for statistical computation and graphics. It consists of
....
WWW: http://www.R-project.org/
```

Other useful options are -D to get the install-message, or -f to get the packing list file. We have seen an example of -f above, here is an example of -D, for a port which has an install-message, and illustrating the use of the -x option for selecting patterns:

```
niobe% pkg_info -D -x postfix
Information for postfix-2.2.9,1:

Install notice:
To enable postfix rcNG startup script please add postfix_enable="YES" in
your rc.conf

If you not need sendmail anymore, please add in your rc.conf:

sendmail_enable="NO"
sendmail_submit_enable="NO"
sendmail_outbound_enable="NO"
sendmail_msp_queue_enable="NO"
...
```

As with dpkg -L, pkg_info -L lists all files in a package, and -g shows files which don't match checksum, so have probably been customized. The -r option shows downward dependencies, while the -R option shows upward dependencies. Finally the -s option shows the size occupied.

```
niobe% pkg_info -r atk-1.10.3_1
Information for atk-1.10.3_1:

Depends on:
Dependency: pkgconfig-0.20
Dependency: perl-5.8.8
Dependency: libiconv-1.9.2_2
Dependency: gettext-0.14.5_2
Dependency: glib-2.8.6_1

niobe% pkg_info -R atk-1.10.3_1
Information for atk-1.10.3_1:

Required by:
abiword-2.4.2
abiword-plugins-2.4.2_1
aiksaurus-gtk-1.2.1_4
at-spi-1.6.6_2
.....

niobe% pkg_info -s atk-1.10.3_1
Information for atk-1.10.3_1:

Package Size:
924     (1K-blocks)
```

So any possible available information on a package can be displayed by pkg_info. This information is obtained by examining the various files under `/var/db/pkg` that we have called the package database. If this set of files has inconsistencies, which may appear for different reasons, the information will be inaccurate. The author has writen a python script check_pkg.py [10], able to check the consistency of the +REQUIRED_BY files, and to update the origins in the +CONTENTS files, doing basically the samething that portupgrade does.

Compared with Debian there is some unavailable information related to the state of a package, simply because this notion does not exist in the FreeBSD system, where either a package is installed or it does not exist. Under Debian, it can be installed, half-installed, not installed, unpacked, half-configured. It keeps state on its selection status, it can be selected for install, deinstall, or purge. Purge is stronger than deinstall, it removes all config files. It keeps also state on flags, it can be marked hold, so that dpkg will not mess with it, or marked reinst-required, meaning that it requires reinstallation, because installation was not successful for some reason.

So Debian has a whole zoology of states for packages, mainly related to the elaborate configuration procedure, where FreeBSD has essentially none. The configuration procedure has a lot of stages which are discussed in the dpkg man page. One obvious drawback is that these configuration steps are very slow, frequently requiring more time than downloading and installing upgrades. Another is that it sometimes requires user interaction, and finally that it may require knowledge from user that he doesn't have and doesn't want to acquire. The question remains to know if this sophistication really serves some useful purpose.

3.3 Sysinstall.

The last tool we mention is sysinstall because it has been written by the same author, and has some interactions with the packages subsystem. It is the FreeBSD installer which is curses based. It has options to install precompiled packages from cdrom or from the network. When doing that it first reads the INDEX file that we are just going to discuss, and is present in the subdirectory "packages" either on cdrom or on the ftp site. From this he gets knowledge of all the available packages, classed by category, and presents them this way to the user. Each package has a short description, the COMMENT allowing the user to understand the purpose of a package. When chosen, all its dependencies are automatically checked, using the dependency information in the INDEX. All this is presented in curses screens which a are quite simple and readable, certainly much more than the corresponding Debian tools dselect, a sure way to get headache, or than aptitude in curses mode. The most similar Debian tool is in fact Synaptic, but with a nice GUI. It is remarkable that this simple and efficient tool has been devised many years before synaptic was introduced.

4 The index file.

The index file is a database of rows, one for each port in the ports system. It lives directly under the portdir /usr/ports, and is called INDEX in FreeBSD-4, INDEX-5 for FreeBSD-5, INDEX-6 for FreeBSD-6, etc. Programmatically the number is obtained by picking the first letter in uname -r. There are differences between these various versions of FreeBSD because there are ports which build on one version but not on another one.

Each row has 13 fields, separated by the pipe symbol — like in the output of make describe. However the fields are not exactly in the same order, the first seven ones are the same, but after that, the less frequently used dependencies, fetch, patch, extract, are relegated to the end, and are frequently empty. Unlike make describe, the build_depends are augmented by the lib_depends, and similarly the run_depends are also augmented by the lib_depends. But the fundamental difference is that these dependencies are recursively expanded (by the run dependencies of first stage dependants) and mapped to package names. So the list of fields is:

1. pkgname
2. path
3. prefix
4. comment
5. descr
6. maintainer
7. categories
8. build_deps
9. run_deps
10. website
11. extract_deps
12. patch_deps
13. fetch_deps

It is easy to extract information from the INDEX file by using grep and awk. For example to obtain the map from package name to port name, one can grep on the first column and pick the second, as in:

```
niobe% grep -e '^atk-1.10.3_1' /usr/ports/INDEX-6|awk -F\| '{print $2}'
/usr/ports/accessibility/atk
```

Clearly we can get this way the same type of information that pkg_info allows to obtain, but this time from the ports tree instead of the package database. A difference is that INDEX contains the downwards dependencies, but not the upwards ones, except indirectly, of course, by scanning the whole INDEX.

Let us remark that, while the INDEX file is potentially very useful as a central database which links the port tree and the packages, it is very little used by the FreeBSD ports system in its standard form. Each time one compiles and installs a port, the dependencies are recomputed using bsd.port.mk magic, which takes time, but this time is masked by the compilation time of the port which is much longer, so nobody notices. The package tools only deal with stuff under /var/db/pkg, except pkg_version, which is very little used. Sysinstall is the main tool which uses the INDEX routinely, but people use sysinstall uniquely for initial installation. One can very well run without any INDEX

file. If one wants to create one, one runs in the directory /usr/ports the command:
`make index`

This basically runs “make describe” in each port directory in the ports tree, and feeds the whole result to a perl script `make_index` living in `/usr/ports/Tools`. When the perl script has the entire content in memory, it can recursively expand run dependencies and build dependencies, and replace all port origins by the corresponding package names. As one may imagine this needs a lot of time, and a lot of memory. It takes of the order of an hour on a recent machine, and of several hours on an old one. This is because ”make describe” takes of the order of 0.1-1.0 second per port and there are 15000 ports. Moreover this number is fast growing, of the order of 100 per month or more. The perl script takes also a lot of ressources. Hence there is a scaling problem, which is not easy to solve, except by inventing more efficient tools to build the INDEX.

4.1 Incremental builds.

A “solution” which has been tried is the incremental index building, introduced by Radim Kolar, as we have already said. The idea is to detect the ports which have been updated since last time the index was built and run “make describe” only in the ports that have some relation with these changed ports. One expects that this cuts drastically on the number of ports in which one needs to run “make”, at least by an order of magnitude if not two, hence reducing the computational time to a more reasonable amount. Unfortunately there may be a lot of reasons why some ports would need index rebuilding besides being listed in the chain of dependencies of an updated port, as we shall see below. R. Kolar was well aware of that and tried to detect ports which include the updated port, new ports, ports which have moved, and so on. But it is difficult to be bug free in this domain, and it is recommended to do a full build from time to time.

Anyways the author removed his work a few months after its introduction, apparently due to some dissents, so it is no more available. Since it is under GPL i can distribute a copy, here [6]. The same idea has been revived as a perl script, which can be found in the port p5-FreeBSD-Portindex, which i have never run and have consequently no experience. The author, Matthew Seaman, says that running an incremental index build needs a couple of minutes. The price to pay with these setups is that you need to maintain state, like a cache of the previous runs of “make describe” or a “cpickle” of the state of the dictionary describing the ports system, for the Python version - apparently portage does the same for Gentoo.

Due to the difficulty of maintaining an up to date INDEX, many people are sloppy and keep an old version which is not in sync with the true state of the ports, which are updated using `cvsup` or `portsnap`, the last tool being much faster. So using the INDEX would mean obtaining and perhaps writing somewhere inconsistent data. To obviate this partially, there is a command `make fetchindex` which fetches a recent, if not up-to-date index directly from FreeBSD web site. An INDEX is of the order of 8 Megs, so this not a small download, but it is not a big one either compared to some mainstream downloads. An up-to-date INDEX is used by the popular freebsd port, “portupgrade” which aims as being a sort of “apt-get” for FreeBSD, and is widely used.

4.2 The readmes.

There is a target `make readmes` which goes to each port directory and writes here a file “README.html”, containing the output of pretty-print-run-depends, pretty-print-build-depends, the name of the package, the short description, a link to the long description, the website, and the maintainer, plus links to navigate the ports tree. All this using templates in /usr/ports/Templates for the accompanying text. It is a nice way to browse the ports tree while displaying the corresponding INDEX content. The target is implemented by running recursive make to each port tree, and here spawn a perl script `make_readmes` in `/usr/ports/Tools` which takes the corresponding INDEX line, splits it in fields, substitutes the templates, etc. Needless to say this procedure takes an long time while it could take a couple of minutes if the perl (python, etc.) script was doing all the job itself, since the required information is already here. Even more simply one can imagine a small python web server (i have written such a server [13]) displaying the same information extracted from INDEX, which would avoid cluttering the ports tree with README.html files, would take a few lines to write and would be amply sufficient to do the job. Anyways the R. Kolar “portindex” has a tool to write these readmes, which is fast.

5 The dependency problem.

As we have already said, the main problem of modern package management systems is to deal with dependencies. The first Linux distributions had no notion of dependency management, for example the first Slackware distributions offered a bunch of packages in the form of tarballs without any metadata. The first RedHat packages were rpms with some metadata but absolutely no tool to automatically download dependencies, and needless to say, no tool to upgrade. Up to now Slackware still has no great tool for package management, RedHat has a pale copy of apt-get, called “yum”, which is very slow, and probably not reliable (Mandrake has a similar tool, urpmi), so many users prefer erasing the disk and reinstalling as the solution to upgrading the system. The first innovative system has been the FreeBSD ports system which downloads dependencies automatically, in a reliable way.

Unfortunately upgrading is not so easy short of erasing the disk, so when the apt-get system from Debian was introduced it was immediately seen as the miracle solution to the “no fuss” management problem, and the model for all other distributions. At present, the most popular Linux distribution, by far, is a Debian derivative, Ubuntu, which offers all the niceties of automatic maintenance via apt-get (or its modernized version, aptitude). In the FreeBSD world, a similar tool has been introduced, named portupgrade. It is an ingenious collection of ruby scripts, which work on top of the infrastructure we have already described, and in particular use internally the pkg_tools (pkg_info, pkg_delete, pkg_install) plus the INDEX file to gain information about the state of packages and ports, and manage packages. I have used it but i cannot say i am really happy with it, and certainly not at the level one can be happy with apt-get.

One of the reasons is portupgrade’s fault, it is extremely slow, paying here the price of using ruby. Last time i upgraded KDE with it, using binary packages, on a modern machine, it took several hours. Another heritage from ruby is the general instability, such as the inability to read databases after upgrades of ruby or the db connector to ruby, which has afflicted people recently, and has required rebuilding of databases. A python version of the same tools would certainly do a great service, offering a much faster performance, and certainly stable use. A great C or C++ version would be fast, like apt, and would nor require extra dependencies.

But the tools themselves, written by Akinory Musha, are nice. The programs are very cleanly written, and contain a lot of nice ideas. The second reason of dissatisfaction has nothing to do with portupgrade itself, it comes straight from the FreeBSD ports tradition of compiling from source, which portupgrade perpetuates blindly. Upgrading a package with its dependencies may easily need the compilation of a large number of ports. There is always the possibility that one of the ports in the chain doesn’t compile, at which point you are screwed. Countless people have ruined otherwise working systems by trying to upgrade an innocent looking port with portupgrade. Even if everything compiles, you have no guarantee that everything works. The only way to ensure that a package management system will not ruin a machine is to **exclusively use binary packages** which have been fully tested as working. If a necessary binary dependency doesn’t exist, for some reason, you upgrade nothing and don’t ruin anything. The single most important reason that apt-get works is that Debian provides binary packages for **all** its software. A secondary reason may be that Debian packages may have more useful metadata coming with them, and perhaps better and more formalized configuration procedures, but i am not very convinced of that.

This does not mean that compiling from source much be banned, but it should be reserved to the packages one really cares about, that form the core of the activity, and which need some special configuration or compilation option. Using a ports tree which is a constantly evolving target, with ten or twenty port changing each day, means absolutely no security for the user. For each FreeBSD release, the release managers freeze the ports tree for a couple of weeks, and only bug fixes are received in the ports tree. Then binary packages are compiled on freshly installed machines for sending to the ftp servers. A useful feature of the system would be that these states of the port tree are tagged to be used as cvsup tags for the people who want a known “relatively stable” state of the ports tree, and leave to the adventurous the task of portup-chasing the ever evolving current tag.

A second problem is all those ports marked FORBIDDEN or the like for good or bad reasons, like security problems which in are irrelevant for people who are not exposed to the outside, licence troubles and all other stupid administrativia. With Debian, the system is comprised of packages which exist and are downloadable. In principle there is no risk that apt-get halts in the middle of an upgrade operation leaving the machine ruined. More, apt-get downloads all packages to your machine before beginning a single upgrade operation. I consider this is a prerequisite for a working package management system. No upgrade operation should begin without having previously built all the necessary packages, even if this involves compiling them in a jail to make sure as not destroying the state of the machine.

5.1 The nature of dependencies.

Dependencies in the ports system may be of two sorts, "explicit" dependencies are recorded through the variables RUN_DEPENDS, etc. that we have defined above (et are frequently created completely implicitely through makefile magics), but there are also more hidden dependencies such as "master ports", inclusion of other ports makefiles and stuff like that. These dependencies don't tend to be recorded in state describing objects such as INDEX or +CONTENT files in the package database, but they plague trying to do incremental builds of the INDEX and perhaps package management. Let us give an example. Consider the port boost-python, whose aim is to introduce bindings between the C++ boost libraries and python using template mechanisms. Here is the Makefile:

```
MASTERDIR= ${.CURDIR}/../../devel/boost
PKGDIR= ${.CURDIR}
PLIST= ${MASTERDIR}/pkg-plist

WITH_PYTHON= yes

.include "${MASTERDIR}/Makefile"
```

So the only effect of building the boost-python port is in fact to build the boost port, but setting the variable WITH_PYTHON=yes. As a consequence if the boost port is changed, the boost-python must be considered modified, even if it is exactly the same. Mechanisms of these sort abound in the ports tree, and lead to undetectable dependencies according to the standard mechanisms. Indeed running "make describe" in the boost-python port yields absolutely no reference to the devel/boost port, only lang/python is mentioned as build and run dependency. This is important, e.g. for incremental INDEX builds, and this is the reason why portindex takes great care of such situations by scanning the Makefiles for the presence of ".include" or of "MASTERDIR".

Portupgrade has even more radical tricks, it runs "make -dd -n" in the port to be scanned, that is it outputs all the directory operations of make in debug mode. Then it greps the successful queries, then those directed to /usr/ports, and finally those who don't go to system directories like /usr/ports/Mk. All those queries go to "real" ports, and so lead one to suspect that there is a hidden dependency to such port. By the way, it is clear that all modifications of system makefiles invalidate the whole port tree, since they are susceptible to modify the dependencies of any port.

It is also important to discuss the time evolution of dependencies, because as we upgrade the ports tree, install new ports or packages, inconsistencies appear between the state which has been recorded on disk and the actual software or port tree, i.e. the dependency situation becomes fuzzy. A particularly obvious case is when some ports disappear, either completely or because they have been moved to another topdir, have changed name, etc. A recent exemple being the change of name of ethereal to wireshark. All users of portupgrade have been greeted from time to time by the message "Missing origin: package" meaning that the port has disappeared. Obviously an automated solution to this problem is not possible and the end user has in general no reason to know the answer to the renaming problem. One possibility is to consult the cvsweb of the ports tree to discover a clue, but an easier solution has been introduced, marking in a file MOVED all such changes. As far as this file is really current and exact, it is possible to guess the new origin. This doesn't solve the problem of ports which have disappeared, like portindex. Another possibility leading to the "duplicated origin" message is that the port system may install two versions of a given software, this happens frequently with "tcl" and "tk", due to an upgrade after some period of time, and both could serve as a dependency requirement for some other programs. Then the automated system is in trouble. To summarize, with the flow of time, upgrade of ports and installs of ports and packages, the coherency of the dependency situation rottens.

Another dependency problem related to time evolution is the fact that the package obtained from a port may depend on software which has been previously installed on the machine. This can happen if the "configure" script has autodetection features and chooses to add features if they are supported on the machine. This is the case, e.g. for "mplayer". As a consequence, between two machines equipped with the same set of packages, the real content of the packages may be different, even the dependencies may be different, according to the temporal order in which these ports have been compiled. It may even be that, after having detected the presence of a feature, the compilation crashes because the corresponding support is buggy. In the same circle of ideas, we have already given the example of gimp, which compiles to the gimp or gimp-gnome package according to the presence of gnome-panel. It is clear that the number of such possibilities is infinite, while with precompiled packages such as those offered in FreeBSD repositories, or in Debian repositories the result is deterministic.

Let us assume that we can neglect all those real problems, and concentrate on the standard dependencies seen by the FreeBSD ports system. For some reason the system introduces as we have seen, six different types of dependencies, fetch, extract, patch, build, lib, run. In all tools, lib is in fact added to run and build, for good reasons: the program will not run without all its shared libraries, and will not even build without them because the final link stage will break. It is of little use to discriminate fetch, extract, patch, because these are clearly build dependencies, you don't need a single of them to run the program, and you need all of them to build it.

Hence there are essentially two types of dependencies, build time dependencies and run time dependencies. This corresponds to the Linux situation where devel packages are almost always separated from ordinary packages, the latter one serving to fulfill run time dependencies, while the former are used for building programs. For the purpose of having a working system, the build time dependencies are totally irrelevant, and the only concept of interest to analyze the dependency relations is run time dependencies. Note also that we have seen that in the recursive expansion of build deps, only the first level direct build dependencies are used, the rest of the recursion using only run deps. This is because for building a given package we need only run times of the tools, not tools to build tools. For example if we want to build a Java program, we may either need gcj and classpath, or the sun-jdk, but nothing forces us to spend hours compiling gcj itself or sun-jdk. Hence in the following we shall limit ourselves strictly to run time dependencies, which is also what the package database does, and what working systems like Debian do.

A concept that Debian has, but i don't think FreeBSD has, is the one of abstract dependencies which can be fulfilled by several alternatives. For example we may have the concept of a "Java development tool", which will be a requirement for running Java applications, but which can be fulfilled either by gcj and classpath or by sun-jdk. In the base system one may have the requirement of a MTA, which can be fulfilled either by sendmail, postfix or exim. This flexibility allows to enforce the presence of a complete chain of dependencies without imposing a specific choice. Anyways, once choices have been made for such abstract dependencies, we are back to the traditional situation such as in the FreeBSD system.

One last remark. Even run time dependencies may be of different sorts. The most traditional and annoying is shared libraries. If a required shared library is absent, the program doesn't run, period. It is completely broken. Other similar dependencies have less dramatic effects, plugins. The absence of a plugin will mean reduced functionality, but we still enjoy part of the functionality. However dependencies may be of other types, for example some data files. An example is a firmware file which is necessary for running some piece of hardware.

Dependencies can even be executables, for example the font editor fontforge may use the bitmap tracing tool autotrace to convert bitmap fonts to postscript outlines and further edit them. Then autotrace works as a coprocess, and without it, there is reduced functionality. Note that this dependency is not recorded by FreeBSD. To summarize, there may be all sorts of run time dependencies, fatal or not fatal, recorded by the system or not, bogus or not, still we have to simplify and talk only of the run time dependencies recorded by the system, whatever they are. Moreover we are speaking of recursively expanded dependencies.

A closely related concept is the concept of **conflicts**. This means that two ports are declared conflicting, you cannot install both, usually because both install files of the same name in the same place. An example is editor/emacs and other emacs variants. The Makefile of the emacs port has:

```
CONFLICTS=      emacs-19.* \
                xemacs-[0-9]* xemacs-devel-[0-9]* \
                xemacs-mule-[0-9]* xemacs-devel-mule-[0-9]*
```

All other emacs variants ports, emacs-19, xemacs, etc. have similar declarations. So this is a case where installing a port, instead of requiring another port, requires its absence. Note that like dependency, which has to be recursively extended, conflicts also imply recursion. Indeed if portA requires recursively portE, and portB requires recursively portF, but portE and portF are conflicting, this implies that portA and portB conflict. Clearly this generates great complexity in a port management system, notably Debian apt system has a lot of consideration for this subject. Fortunately it seems that FreeBSD don't have a lot of CONFLICTS declarations, so this seems a minor problem here.

Finally the dependency mechanism is used to compose metaports whose only aim is to bring a lot of dependencies. For example the port x11/kde3 is useful only to bring all optional KDE components. Moreover this is parametrized by variables in the following way:

```
.if !defined(WITHOUT_KOFFICE)
RUN_DEPENDS+=   kword:${PORTSDIR}/editors/koffice-kde3
.endif
```

which will bring koffice if WITHOUT_KOFFICE is not set. A more trivial example is the port misc/instant-workstation which has:

```
RUN_DEPENDS=    acroread7:${PORTSDIR}/print/acroread7 \
                bash:${PORTSDIR}/shells/bash2 \
                cdrecord:${PORTSDIR}/sysutils/cdrtools \
                dos2unix:${PORTSDIR}/converters/unix2dos \
                emacs:${PORTSDIR}/editors/emacs20 \
  ....
```

that is the list of desired ports.

5.2 The dependency DAG.

Given two packages, packageA and packageB, either packageA depends on packageB (package A requires packageB, or packageB, "is required by" packageA), or conversely packageB depends on packageA, or finally, packageA and packageB have no dependency relation, i.e. dependency is a partial relation. If packageA depends on packageB and packageB depends on packageC, then packageA depends on packageC because we have assumed that dependencies have been recursively extended, in particular those of packageA. So the relation is transitive.

Finally if packageA depends on packageB we must install packageB before packageA. If packageB depends furthermore on packageA, we must install in the other order, so there is a deadlock situation. Hence it is necessary to ensure that this only occurs when packageA is equal to packageB, that is reflexivity. We have seen that the dependency relation is a partial order relation. In such cases it is frequently useful to extend this order relation to a **total** order relation, that is, enumerating all packages in some order such that if packageA comes before packageB, then we know that packageA depends on packageB. So when we have a set of packages, we know we can safely install them beginning by the end, the smaller with respect with this order relation, and progress all the way to the first. Similar for upgrades.

This extension problem is frequently presented in graphical form. Imagine a set of nodes, with a node for each port in the system, and a set of arrows, going from node to node, where there is an arrow from nodeA to nodeB if corresponding packageA depends on packageB. This forms a directed graph, directed because the edges are oriented by the arrow heads. Now the fundamental property of this graph is that it doesn't have any **cycles**, at least it should not have any. This means that there is no set of nodes, N1,...,Nk such that there is an arrow from N1 to N2, from N2 to N3, etc. and from Nk to N1. This does **not** mean there is no unoriented cycles in the graph, i.e. it does not mean that the graph is a tree. The reason is the same as for the reflexivity of dependency relation, this would produce a deadlock in the installation procedure of packages, and is thus expressly forbidden in the Porter's handbook. In other words, it is the responsibility of port maintainers that there are no oriented cycles in the graph. It is also a direct consequence of transitivity and reflexivity, since under the above hypothesis, N1 is greater or equal to Nk and Nk greater or equal to N1, hence they are equal and the cycle has one less node.

Such a graph is called a **directed acyclic graph** (DAG). The problem of extending the dependency relation to total order consists in setting all the nodes on a line in such order that all arrows go from left to right. This is called "topological order sorting", and there is a program /usr/bin/tsort in the base system which does that. Of course this sorting is not unique, but we don't care. Let N be the number of nodes and V be the number of arrows. There is a simple algorithm of order (N+V) which produces a topological sort, due to Tarjan. It consists of first scanning the graph, noting all nodes, the list of arrows pointing to them and the number of arrows pointing from them, also noting all arrows, and keeping track of the couple of nodes between them. We also maintain a stack of the nodes with no arrow originating from them. This is clearly O(N+V). Then one chooses a node in the stack (the liberty of choice here exactly describes all possible orders), we decree it is the smallest element of the graph, remove it from the graph. We also remove all arrows pointing to it from the set of arrows, and decrease by one the arrow count of their origins. All the nodes whose arrow count fall to 0 (so they were previously of level 1) are added to the stack. If we want to maintain a natural level ordering we can push them to the beginning of the stack, while we pop from the end. We are now in the same situation as in the beginning but with one less node. In N steps we are done. Since we have at most V arrow deletions to do globally, this recursive procedure requires O(N+V) operations in total. Of course one needs to be careful programming it, not introducing stupid O(NV) or similar procedures, since the numbers here are very big. When done properly this algorithm is really minimal with respect to the number of operations being done.

Loop detection is also built in. Suppose that at some point in the algorithm the stack is empty but the graph is not. This is because we have a loop, all elements of which have arrow count one, and we cannot find any element with arrow count 0. Hence the algorithm fails if and only if we have a loop. To proceed we need to arbitrarily remove an arrow (that is remove a dependency) which breaks the cycle. In theory there are optimal breakings, but here this optimality means nothing because only human intervention can discriminate the correct dependency to remove. Hence the only meaningful solution is to display the cycle for further study, which is trivial at this point. The program Portupgrade has such a topological sorting procedure built in, in the ruby script pkgtsort.rb. Recall it is necessary to start upgrading a set of packages in correct dependency order.

5.3 Portupgrade.

Portupgrade consists in a collection of executable ruby scripts, portupgrade, portcvsweb, ports_glob, portsclean, portsdb, pkgdb, portversion, pkg_deinstall, pkg_fetch, pkg_glob, pkg_sort, pkg_which, building on a collection of ruby library packages, pkginfo.rb, pkgtsort.rb, ports.rb, pkg.rb, pkgmisc.rb, pkgversion.rb, portsdb.rb,pkgdb.rb, pkgtools.rb, portinfo.rb, and of course internally on the standard pkg_tools. All of them are due to Akinori Musha. The library packages are located in the standard ruby library location. Each of executables has a man page.

Portupgrade maintains two databases, in fact Bekeley-db databases, one INDEX.db (or INDEX-5.db, INDEX-6.db) for ports in /usr/ports, another one pkgdb.db for packages in /var/db/pkg. The command portsdb -u is used to update INDEX.db, while pkgdb -u creates pkgdb.db. Other options of pkgdb allow to query the ports database, for example portsdb -r “port” lists ports depending on “port”, while portsdb -R “port” lists ports required by “port”. Moreover port can be specified in different ways, such as a glob, like in ports_glob.

```
niobe% ports_glob lang/oc'*'
lang/ocaml-doc
lang/ocaml-nox11
lang/ocaml
lang/ocaml-mode.el
niobe% ports_glob atk
accessibility/atk
niobe% portsdb -R atk
converters/libiconv
devel/libtool15
lang/perl5.8
devel/gettext
devel/pkgconfig
devel/glib20
accessibility/atk
niobe% time portsdb -r atk
x11-toolkits/gtk20
math/calcoo
graphics/libexif-gtk
x11/gnome-clipboard-daemon
...
graphics/ocaml-images
portsdb -r atk  37,16s user 19,60s system 99% cpu 56,757 total
```

This illustrates that, first -r and -R options are **reversed** from what we have previously seen, then the slowness of the program, which takes of the order of one minute to find all programs which depend on atk, while we are running on a powerful machine P4 3Ghz, with 1Gig memory, and also the fact that the number of such upwards dependencies can be considerable, here one gets 1041 programs in the port tree depending on atk. Finally we see that globs can be either something like atk or something like lang/'oc*'. A remarkable feature is that portsdb can list master ports, even recursively if -R is specified:

```
niobe% portsdb -M boost-python
devel/boost
devel/boost-python
niobe% portsdb -M -R boost-python
lang/python
devel/boost
devel/boost-python
```

Hence we are perfectly equipped to determine all sorts of dependencies in the ports tree. Also note some size information, INDEX-6 occupies 7.5Megs, INDEX-6.db occupies

15Megs! Moreover portsdb has some convenience features like the -F option which downloads an INDEX file such as make fetchindex.

The program pkgdb does the same job, but for the package "database". In fact pkgdb.db indexes all the files installed in the ports system! No wonders it weights 37Megs on my machine which has 600 ports installed. Anyways it allows to use pkgdb to know from which package a file has been installed, or with the -c option if files installed by "package" have been overwritten by another one. With the -o option, it gives the origin of a package, that is the port which, when compiled, will produce the package, as listed in +COMMENTS. The command pkg_which does exactly the same thing, except it can search a given file in PATH.

```
niobe% pkgdb /usr/local/lib/libatk-1.0.so.0
atk-1.10.3_1
niobe% pkgdb -o atk-1.10.3_1
accessibility/atk
niobe% pkg_which display
ImageMagick-6.2.5.5_3
```

But the most important, central and controversial role of pkgdb is the functionality provided by the -F option, interactive fixing of the package database. In fact one of the fundamental ideas of portupgrade is that, as we have explained, the standard package database suffers bit rot as new ports or packages are installed, and that it needs to be fixed. Hence pkgdb -F proceeds to fix all the information in the files sitting under /var/db/pkg, notably the dependencies appearing in the +CONTENTS files, and in the +REQUIRED_BY files. This is controversial because once these modifications are done, the previous state is lost forever, while modifications to some private database would be safe. The problem being that it is highly probable that these modifications are bogus, all the more when user is prompted to help fix inconsistencies, while he is usually the less able of having any clue about the necessary modifications. A package management system should never assume that the final user knows anything, or wants to know anything about the complex net of dependencies in his system. He wants to see a black box, which works. Citing the man pages, "pkgdb helps resolve stale dependencies, unlink cyclic dependencies, complete stale or missing origins and remove duplicates. You should run this command periodically so portupgrade(1) and other pkg_* tools can work effectively and reliably".

Recall that stale dependencies are ports which were present when the package was installed and on which the given package depends, but have since then disappeared and don't show up in the ports tree. It may be because they have completely disappeared like portindex, or because they have moved, either or name, or changed place in the ports tree. Portupgrade may reassociate automatically such dependencies to the correct new location if the file MOVED contains the necessary information. Stale or missing origins represent the same problem, a package whose port has disappeared or been moved. Breaking dependency cycles is the problem we have mentioned above, about DAGs. In principle there should be no cycle, if there is one, only user intervention can choose the correct dependency to remove which unbreaks the cycle. Finally if we have a port which needs, say tk-8.4.3 and we install afterwards a port which requires tk-8.4.11, the first port will be able to resolve its dependencies by either version of tk. Hence portupgrade will prompt to remove the dependency to tk-8.4.3 and will subsequently rectify all dependency records mentioning the old version, to the new version.

The program pkg_sort allows to see topological sorting at work.

```
rose% pwd
/var/db/pkg
rose% \ls | pkg_sort
No such package is installed: XFree86-4.3.0,1
No such package is installed: pkgdb.db
No such package is installed: portindex-18_1
ispell-3.2.06_2
libtool-1.3.5_1
uulib-0.5.18
lrzsz-0.12.20
unzip-5.50
...
kdeaccessibility-3.3.2
fr-kde-i18n-3.1.1_1
gtk-2.2.1
firefox-0.8_4
kdeutils-3.3.2
```

This shows names which don't correspond any more to something in the port tree or are not ports, then packages with no dependencies are listed, at the end are packages with maximal number of dependencies. Of course they should be upgraded in order. The program portversion is the ruby version of pkg_version, but using the Berkeley databases, which speeds it up. It shows packages which need to be updated according to the ports.

We have now described the building blocks in the operation of the main program in the system, portupgrade. First assume one has run pkgdb -F so that all dependencies are correct. Afterwards each operation will update pkgdb.db so one does not have to worry about it, in principle. If used with the -N options it is equivalent to portinstall, and installs a new package. Otherwise, it upgrades packages. Packages can be specified as in pkg_glob, so it may be a collection of packages. First the collection is topologically sorted, then one proceeds to upgrade packages one by one beginning by the less dependent ones. The procedure is to backup the package (the package can be kept using the -b option) install the new package, register the new package in the database, and interate. All those operations are done using inferior commands in the standard pkg_tools set, such as pkg_create -b, pkg_delete, pkg_add. Between each operation portupgrade checks the consistency of its databases, which is very slow. One can make the procedure speedier but certainly not speedy by setting the -O option which omits these checks, assuming that pkgdb -F has been run and has done a good job. With the -p option, copies of the freshly built packages are kept. Assuming that everything works OK, when the new package is installed and registered, portupgrade deletes the copy of the backup. If something gets wrong, it reinstalls the old package. This of course doesn't save us from a partially upgraded machine. Some times portupgrade is picky on the version number of dependencies, so it is necessary to set the -k option to keep it going. We enter here the cases where things are beginning to screw up.

The new version of the package is obtained by looking at the origin of the old one, and looking in INDEX.db the last version of the corresponding port. Of course one needs a bit of code to parse the version, revision, and epoch numbers, in order to decide which version is newer. Moreover the fact that a package is more recent than another one may be seen on the prefix, for example py24-tkinter compared to py23-tkinter, which further complicates the situation, since parsing names is not trivial. In general the new version will be obtained by building in the port tree.

Let us mention portupgrade has a configuration file `pkgtools.conf` in /usr/local/etc which is in ruby syntax. In particular one can specify here variables which will be passed to the make command running on behalf of portupgrade. They are kept in a ruby dictionary in the following form:

```
MAKE_ARGS = {
    'databases/mysql323-*' => 'WITH_CHARSET=ujis',
    'ruby18-*' => 'RUBY_VER=1.8',
}
```

so the compilation of a ruby18-* port will be launched as make RUBY_VER=1.8. This acts somewhat like /etc/make.conf, except that it seems that secondary makes directly launched by the first one will not see the variable. Assuming all goes well the port is compiled and installed, as well as its dependencies and all goes well. A nice feature of portupgrade is that it can automatically upgrade not only a port, but its dependencies, by setting the -R option. For example, to update all KDE stuff which has been installed as dependencies of the metaport kde3, it is sufficient to issue portupgrade -R kde3. To follow in the confusing state of affairs about this notation, the man page calls these dependencies upward-recursive, while they are clearly downward. The author must be thinking in terms of reversed arrows, or is basing his notion of dependency on the +REQUIRED_BY files. When the -R option is present, the set of packages that portupgrade considers will be dependency complete, so portupgrade will begin by the least dependent ports, hence the "make" issued in these ports should never issue secondary makes that we discussed just above. So in this case the make arguments set in pkgtools.conf should be obeyed precisely.

Using the -r option, it is the set of packages which depend on the given package which are upgraded, that is the packages listed in +REQUIRED_BY. If we upgrade a basic library like atk, it is reasonable to upgrade all those packages which use the shared library libatk, which means half the packages on the machine, unfortunately. However one of the **very nice features** of portupgrade, is that, when deinstalling a package it **preserves the shared libraries** it contains. They are obtained by running the program “file” on all files listed in the package content, and grepping lines which contain “shared object”. Portupgrade (or pkg_deinstall) stores them in a central place `/usr/local/lib/pkg`. It then runs ldconfig -m on this directory so that they are visible. In most cases this is sufficient to ensure that the packages dependent on our given package will not break, even if we don't upgrade them. Some time later, one can make a more complete upgrade and purge this directory, after having checked that nothing breaks (for example moving it elsewhere). If one is willing to live dangerously, the -u option inhibits this security.

If one does not desire to compile ports, and wants to use packages, then one can set the -P option, which will use packages wherever possible and compile ports otherwise. There is also the -PP option wich uses packages only. However the algorithm to determine what package version to use with -PP is not the same as with -P, so portupgrade will be more laxist if the -PP is present, and will install packages whose version number is not optimal as long as they exist. With the -P option it will be more compilation prone, even when packages exist. Finally the choice of ports of packages may be more fine grained, by filling two ruby arrays in pkgtools.conf, USE_PKGS and USE_PKGS_ONLY. For example, USE_PKGS = ['perl', 'python']. Filling such lists when you have hundreds of ports is not very realistic. As in Debian there is a possibility of keeping packages on hold, that is, portupgrade will not upgrade them, by filling the ruby array HOLD_PKGS. In other words, filling pkgtools.conf offers a **lot** of configurability, and ensures that these settings will be kept forever afterwards. However it requires some work and some knowledge of portupgrade.

We must explain how packages are searched. By default, they are searched on main FreeBSD site, in the subdirectory where the packages corresponding to the actual FreeBSD version (uname -r) of the machine are kept. Typically something like pub/FreeBSD/releases/i386/6.1-RELEASE/packages under ftp://ftp.freebsd.org/. Here are two subdirectories All and Latest. All the packages are under All. The other subdirectories correspond to **categories** and contain links to packages in All. Latest also contain links to packages in All but with names stripped of version numbers. For example it contains the package atk.tgz, which is a link to ../All/atk-1.10.3_1.tbz. (where .tbz means .tar.bz2). In fact portupgrade tries to fetch (using inferior command “fetch”, hence man fetch is useful, notably to use proxies) the package in Latest. One can specify other locations for packages by using environment variables. For example, if one has two cdrom readers, with the two cdroms of a FreeBSD release, mounted as /cdrom1 and /cdrom2, one can specify:
```
PKG_PATH=/cdrom1/packages/All:/cdrom2/packages/All
```
and portupgrade will search compiled packages here. This solves the famous problem of continually exchanging cdroms. If one wants to get them from the web, it is the

appropriate environment variable for “fetch” which needs to be modified, for example `PACKAGEROOT ftp://ftp.fr.freebsd.org` will query the French mirror. This works also with the pkg_add -r command. One can also set PKG_SITES in pkgtools.conf.

For the extra cautious, there is the possibility of running portupgrade while listing results in a file, the -l option, and to only fetch necessary packages, the -F option. This should ensure that there is no problem or help identifies problems and see wether they are dangerous before breaking anything in the machine. For the extra impatient there is the -a option which upgrades everything in one stroke, and the -f option to force action to occur so as to destroy the machine faster.

Another way to proceed is to deinstall packages until one gets to a small number of ports, run pkgdb -F to be sure everything is OK, and then reinstall what was removed with portinstall, preferably using big globs and metapackages. To remove desired packages, there is the program pkg_deinstall which is very convenient because it has recursive options -r and -R like portupgrade and can preserve shared libraries, so minimizing the risks that would be very present with the basic tool pkg_delete. Going to a minimal clean state instead of upgrading port by port is in my opinion, more secure and far faster. I have recently run portupgrade -O -PP -R kde3 on a reasonably powerful machine, with all packages locally present on cdrom, it took several hours. A convenient feature would be that pkg_deinstall emits a list of ports, with maximal number of dependencies, suitable to restoring the machine in its initial state. This procedure has also been the procedure recommended by the Gnome team to upgrade Gnome, because rebuilding Gnome under un unclean state may very easily break.

5.4 Portmaster.

Portmaster is a new system introduced recently by the well known FreeBSD developer Doug Barton. Its main peculiarity is that it is a single shell script, but of enormous proportion, around 1500 lines. This has the advanatge of not requiring any extra tool. It is notable that it is extremely well written and readable, and obviously an example to all shell writers. It is entirely centered about maintaining the package sytem from compilation of ports, so will not interest people wanting to use precompiled packages.

One of the aims of the author was getting rid of any dependency on state of the packages system besides the one contained in the filesystem oriented package database. So it doesn’t rely in any way on the INDEX, nor on any db oriented tool like portupgrade. For any given port, it will follow the “MOVED” file to discover if it has taken a new origin, it will recompute the dependencies using “make build-depend-list” and similar tools and will use recursive subshells to do similar treatment on dependencies.

The script is extremely careful in running make configure, make fetch, etc. separately and logging everything happening, so it is certainly very useful for people who don’t have a lot of ports and want to keep them current frequently. In the course of doing its job it will discover incorrect entries in the package database and fix them more or less in the same way as portupgrade. I am not convinced this is a very useful idea, rather than a procedure susceptible to completley screw the package database to the point it looses the little usefulness it has.

Clearly all those recursive shell invocations, checks of all sorts, etc. consume an awful lot of time, but it is completely screened by the compilation time of ports, so portmaster can be seen as an efficient tool for upgrading a machine through compilation of ports. Another peculiarity of portmaster is that it takes great pain of keeping track of distfiles and their freshness, apparently this is a question which concerns some people. Another notable feature is that it keeps tracks of ports on hold by looking at +IGNOREME files in the package database (one of the two ways of doing so in portupgrade).

5.5 Pkgupgrade.

The author has writen a python program, pkgupgrade [11], to help solve what he sees as issues in portupgrade, while keeping some of its advantages. The first idea is not using any state keeping mechanism, so as to not rely on databases, etc. In this, it is very similar to portmaster and recomputes all necessary dependencies. In particular pkgupgrade doesn't rely on the indications in the package "database" which we have explained have a natural tendency to suffer bitrot. Conversely it doesn't try, contrary to portupgrade, to fix them, it simply ignores everything except the installed package names and their origins that it extracts from the relevant line in +CONTENTS. However, unlike portmaster, its aim is to use precompiled packages as far as possible, hence the name pkgupgrade. The aim of this program is very different from the aim of portupgrade or portmaster, it is intended to be used infrequently, with massive upgrades, and the majority of packages being precompiled, basically just after a FreeBSD release.

Let us repeat that the preference given to precompiled packages is motivated by the desire of having *reliable* upgrades, since there is always a non negligible probability that a compilation fails, for example because the distfile has disappeared. A secondary motivation is that some ports are extremely long to compile, particularly big C++ frameworks such as KDE, OpenOffice, etc. There is no gain whatsoever to be expected from recompilation on ones machine of such big software. The situation is very different for server software installed on server machines, where the administrator may have chosen to use particular Makefile settings, best suited to his situation, and where moreover frequent security upgrades are necessary. The author does not advocate the use of pkgupgrade in such situation, portupgrade or portmaster are very well suited for maintaining such installations. However one may demand that these particular ports are always compiled while the rest is installed from binary packages by listing them in COMPILE. This combines the reliability and speed of pkgupgrade with the flexibility of being able to compile ones favorite ports.

As a consequence, one of the aims is running as fast as possible. On the other hand one assumes that disk space is cheap, that people don't have objections to having so–called bloatware like python on their machine, that an Internet connection is available, and there is no problem downloading a lot of packages. This is not a program for minimalistic people. Note that using compilation from source requires downloading a similar quantity of distfiles. However, contrary to portupgrade, which uses several ports besides the ruby port, pkgupgrade uses exclusively facilities included in the standard python port, without any addition. It has been developed with python-2.4 and will not need any change for python-2.5.

A prerequisite is having updated the ports collection to a state at least as recent as the release one intends to update to. One possibility is to simply extract the ports.tgz from the RELEASE cdrom, as well as using the packages in the same cdrom. This ensures that the ports state is exactly coherent with the packages state, which can only minimize problems. A more recent state of the ports tree should make no difference, except that the ports we will compile will be more recent, and that some ports may have been removed in between, which in principle pkgupgrade deals with gracefully. It may also happen that the dependency relations are not the same in the ports system and in the RELEASE packages, so that minimizing the distance between the state of ports and binary packages is always a good idea. Since one cannot rely entirely on packages and some ports will necessarily be compiled, it would not be a solution to disregard the port tree and base the dependency analysis on the INDEX for packages.

The program proceeds in the following way: first one determines all installed packages and finds their origins. Then one follows the "MOVED" file to discover if this origin has moved or the port has disappeared. Doing so one obtains a list of origins covering most of the installed ports. Then one runs appropriate make -V commands for each port in order to discover its run-time dependencies, and adds appropriate origins so that the procedure closes under dependency. Simultaneously one downloads the latest INDEX from a FreeBSD ftp site corresponding to the appropriate RELEASE, e.g. FreeBSD-6.2 if uname says so. Here one locates all precompiled packages that could be installed to fulfill the above requirements. The other ones will need to be compiled, so the script extends its analysis to build-time dependencies for these ones, and also tries to find precompiled packages. Optionally one can replace a RELEASE distribution of packages by "Latest" ones, but then the coherency is less guaranteed.

This gives a list of ports which is closed under dependencies, and for which we build a complete INDEX. This is rather long, so we use threading to achieve maximum parallelism. As a by product we are also able to get the list sorted in topological order. Moreover we know which packages are sufficiently up to date, which ones will need to be upgraded by binary packages, and which ones will be compiled. One can put some ports on hold, they will not be considered at all, by putting them in a HOLD list. Similarly there is a list COMPILE for ports which should only be compiled, never installed from binary packages. This accommodates the needs of people wanting to tweak the build parameters of particular software, by inserting appropriate options in /etc/make.conf. Note that dependencies of such ports will still be installed from binary packages if possible.

Each port is named after the last name it gets following the MOVED file, or the last valid name before removal in the same file. Precompiled and installed packages are coerced to use the same naming scheme, so that we may compare them. The strategy is to consider only the installed packages which have more recent versions, either in binary form, or to be compiled. If a precompiled package exists it will be preferred systematically, even if a more recent port exists. Of course an installed package is never downgraded to an inferior or equal version.

This leads us to a small discussion of the way in which package versioning works. We have already explained that the versioning has several components, the version proper (which usually comes from upstream), the portrevision, which has to do with revisions of the FreeBSD port itself, and the portepoch, which is a knob added to solve version misorderings. We want to add here some comments on the version itself, which, usually coming from the software author, suffers from severe inhomogeneities which renders determination of correct ordering difficult. Usual version numbers are dot separated numbers like 1.2.3 which are easy to parse.

However difficulties occur when mentions like ”release candidate”, ”alpha”, ”beta”, ”patchlevel” creep into the version. For example it is clear that we want 1.0rc2 < 1.0 because we want the second release candidate to come before the final version. But we also want that a version with a high patchlevel comes after the initial version, perhaps 1.0pl9 > 1.0. In FreeBSD the “official” order is determined by running `pkg_version -t` on two version strings. The code for that is in `version.c`, see in `src/usr.sbin/pkg_install/lib`, and contains a large number of special cases. In pkgupgrade, there is a python routine to determine such ordering, and in case it doesn't work on some nasty version string, there is the possibility of using pkg_version. One should be aware, however, that forking external programs in this way has a high performance cost, for example i have measured around 6s for 1000 executions of pkg_version on a very high performance machine, while the python code runs 200 times faster.

The end result of this analysis phase is a table listing, for the relevant ports (those requiring attention), installed package, binary package existing in the repository, and package which would be compiled from the ports, that is, a priori three different packages. All of them are indexed as explained above by the most recent origin. This represents the ”general case” but there can be different situations. First an installed package may have been removed after installation and before package creation, or after package creation and before the present state of ports (in this case we keep it). Conversely a port may be absent of the installation, but required as a dependency by more recent versions of the software, and thus either present as a precompiled package or to be built.

Finally in case some port has to be built, we also try to install precompiled packages for the build dependencies of this port. This is because it may happen that such build dependencies are extremely heavy to compile, such as a new version of gcc or of java, so that using packages represents an enormous gain. We hope this brief summary gives an idea of the complexity of the situation, having to cope with three different states of the software, two sorts of dependencies (run time and build time), two species of software (precompiled or to be compiled), very little normalization of software naming and versioning, is a lot more than a software like Debian apt-get has to consider.

In a second step, we still use threading to do simultaneously two tasks. We backup all old packages that will be removed, and download from the ftp site all packages that will be installed. To gain space and time the backup is limited to shared libraries and configuration files, and already present backups will not be redone - a useful feature if we run the script several times, e.g. on several machines, with shared backup directory. However packages which will end up completely removed, because they have been removed in the MOVED file, will be completely backed up, and their name will be prefixed by "REM-", so as to spot them easily in case of need.

Shared libraries are detected by running the command "file". The backup itself is performed by a python script, save_pkg.py[12] written by Cyrille Szymanski. It uses some heuristics to speed up the task, such as looking at certain patterns in the path of the file or its suffix. For example it will classify as configuration files, files whose path contain etc, conf, ... or end up in .conf or .cfg. All backups are logged to BackupLog. These backups are kept to help fixing old programs who broke during the upgrade by losing some shared library dependencies, or configuring some new programs where the config files would have been inadvertently overwritten. They have vocation to be erased after some time. It is quite probable that they will not be used at all, this being the author's experience.

To help speed up downloading one can mount a cdrom, typically the second cdrom of a FreeBSD release, the script will look in /cdrom/packages/All to find necessary packages, and will establish a symlink if they are here. Similarly one can use a repository from some NFS share or whatever. The script will only download by ftp the packages it has not found locally. The author has found that, once a first machine has been upgraded, and corresponding backups and downloads done, there is very little to do for the upgrade of a second machine when packages are kept on an NFS share. Usually pkgupgrade will run in less than 5 minutes.

When this is done the script has exact knowledge of present precompiled ports, it proceeds to write a simple shell script whose aim is to remove all packages flagged for upgrade (in inverse topological sort so as to not trigger complaints from pkg_delete), add all precompiled packages in direct sort order (to keep pkg_add happy), finally launch compilation of the ports that we need to compile. This last step is of course susceptible of erroring out, as with all other source based methods, like portupgrade or portmaster. In my experience, even with a recent release like FreeBSD-6.2-RELEASE, there are important ports which have problems, for example mplayer, one of the most sought after ports doesn't install without tweaks on a fresh 6.2 box due to problems with win32 plugins. So it is my opinion that relying on source compilation for building a reliable system is a recipe for disaster. The less one compiles ports, the more the probability of a successful upgrade augments.

To run the program, the best way is to change to a clean directory, perhaps on an NFS share if several machines need upgrading, and simply run it. There are no options, to keep it as simple as possible. There are however configurable settings at the beginning of pkgupgrade. There is no need to run it as root and absolutely no destructive action occurs. Efforts have been applied so that it is as fast as possible, so one may expect to get the results in short time. Of course if there are a lot of backups to do and downloads to perform, it is necessary to wait until they are finished. Otherwise, with a fast machine and a fast Internet connection, the time will be of the order of a quarter of an hour or less, if one has previously mounted a FreeBSD cdrom, even for a large set of installed packages. A lot of the events encountered are logged to UpgradeLog, which may be helpful in case of problems. All downloaded packages end up in directory Packages, and backups end up in directory Backups. The individual files saved in backups can be seen in BackupLog. Of course it is necessary that enough disk space exists to store Backups and Packages.

The main products of pkgupgrade are an index of all installed ports and all their dependencies, named INDEX.ports, (the index downloaded by ftp is kept as INDEX.ftp) which can be explored with a tool like show_index.py [13], and a shell script UpgradeShell, which contains the upgrade instructions. This script can be reviewed at leisure in case one fears some problem. The script is extremely simple and readable, and there is no problem editing it further if it may prove useful. Using a shell script has the advantage that it will run completely independently of any infrastructure provided by the ports system, and will be immune to all package removals. Since no complicated logic is required at this stage, it is perfectly adequate to the job. The shell script has knowledge of the location of Packages, etc. so can be run from anywhere, and preferably from a base system shell on console. When compiling ports it will log success or error in UpgradeLog, so that, at the end of the procedure, one gets a complete log of events in this file.

To run the shell script, of course, one needs to be root, and it will have destructive action. Here there is no difference with other tools like portupgrade or portmaster. However we know in advance what will be removed and can be confident on the packages that will be installed. The elements of risk are localized in the ports that will be built, and will perhaps require user attention, if only to fix configuration. All package removals and installations will proceed at full speed, avoiding to spoil user time. In the author's experience, it took around 2 hours to run UpgradeShell with around 500 ports removed and reinstalled, out of a total of 700. Only 7 ports needed compilation, and 3 compilations failed, by lack of distfiles. This was easily cured by upgrade of the ports tree afterwards. As intended, all deinstallation (15 minutes) and reinstallation of binary packages went smoothly and without any user interaction. One may find that 1h45 is a lot for the installation of 500 packages, but this is the problem of pkg_add. No package management system will be able to run pkg_add faster.

Moreover port compilations will occur on a clean machine, without the clutter of old installations, which i think reduces the risk of misbehavior. In this pkgupgrade differs a lot from portupgrade or portmaster, it is much closer to the ”wipe everything and reinstall” strategy which, in my experience works much better than progressive upgrading, be it on FreeBSD or on Linux distributions. Failed ports will also be logged to UpgradeLog so that one may redo the builds at leisure afterwards. Extensive logging and the backup strategy should reduce risks to the minimum.

5.6 Upgrading.

As a conclusion, portupgrade is a very complete, sophisticated and powerful system to upgrade packages. It can in principle do most of that the Debian apt-get program does. Its defects are, first, that it has so many features that it needs too much knowledge from the end user, compared to the ultimate simplicity of apt-get. And second, that it is of abysmal slowness. Only if this performance problem is solved, one way or the other, it may be a credible alternative to apt-get. The other problems are not really portupgrade problems, but problems of the ports infrastructure. Let us skip the question of the quality of individual ports. This is a problem which can be seen in any distribution whatsoever. If a port maintainer includes an unnecessary dependency, or a completely unreasonable dependency, it will afflict everyone. If distfiles are unfetchable, it will kill upgrades. A ports system can only work correctly with the love of a huge number of maintainers.

But let us concentrate on structural facts. There is nothing systematic to identify what are **configuration files**, how they are preserved between upgrades. Usually configuration files survive these upgrades but it is presumably through the attention of the port maintainer, much more than the existence of solid safeguards. This is a place where Debian is very strong. A problem of this sort that has afflicted users of the “inn” port is that, indeed configuration files under /usr/local/news/etc were indeed untouched, but the “active” file, living under news/db which lists all newsgoups on the machine were zeroed out. Being able to specify that a file has to be preserved at all costs, whatever its location would seem necessary. Another problem with inn is that it depends on a shared library perl.so which is in a completely non standard place. Upgrade perl, and inn is broken, the shared library has flewn away. Since the inn binary is under the non standard place news/bin, nobody will ever run ldd on it to discover the funny perl.so in a dark corner of /usr/local. Such things should be specified somewhere to avoid problems. Similarly for startup scripts under rc.d, but these are being standardized presently.

Another major problem in the FreeBSD system is the constant renaming of ports or creation of new categories, move of ports between them, etc. Choosing a name and keeping to it is the way to go if one wants to have a chance of getting a working system. Introducing prefixes like py24-tkinter is the sort of idea which complicates things with a dubious benefit . If you want to parse such a name, is it prefix plus port name or port name plus suffix. Introducing mechanisms such as gimp becoming gimp-gnome without any user intervention, that is building packages wildly different depending on the state of the machine is still another guaranteed headache mechanism. Non standard variables in /etc/make.conf render all builds non standard. This renders a deterministic upgrade mechanism impossible.

The complexity of the makefile renders its comprehension and cleaning difficult and hazardous. However, in spite of all these problems the ports system is an impressive collection of cooperating processes which seems to live of an autonomous life. It works and works remarkably well. It is able to produce **up to date** packages for consumption by end users in huge number, and with a limited workforce. Compared to the years it took Debian to produce the Sarge release, this is a considerable achievement. This should not mask the fact that the ports infrastructure should be cleaned, in the spirit of pkgsrc (but i am not convinced that splitting bsd.port.mk in a lot of files is the way to go to improve efficiency), some infrastructure should be in place to deal with package configuration, in the Debian spirit, and portupgrade should be much simpler and faster.

5.7 Libchk.

Libchk is a port, "sysutils/libchk", which is still another ruby script written by Akinory Musha implementing the well known idea of following the ldd dynamic dependencies to discover directly what shared libraries are used by a binary, and what binaries load a shared library. This is another way to discover dependencies, admittedly of a particular form, and to avoid disaster in upgrades. This method has been used by the upgrade tool "swaret" for Slackware. It has the advantage that it works automatically, reducing the risks of human error (of port maintainers), but it doesn't consider other dependencies than pure shared libraries ones, and it will not discover binaries located in non standard places, like the inn binaries mentioned above, and thus will not follow their library dependencies. Second, a more treacherous problem is that ldd may fail to discover shared library dependencies of shared libraries in non standard places:

```
niobe% cd /usr/X11R6/lib/mozilla/
niobe% ldd libmozjs.so
libmozjs.so:
ldd: libmozjs.so: Shared object "libmozjs.so" not found, required by "ldd"
niobe% export LD_LIBRARY_PATH=/usr/X11R6/lib/mozilla/
niobe% ldd libmozjs.so
libmozjs.so:
        libm.so.4 => /lib/libm.so.4 (0x281d0000)
        libplds4.so.1 => /usr/local/lib/libplds4.so.1 (0x281e6000)
        libplc4.so.1 => /usr/local/lib/libplc4.so.1 (0x2820d000)
        libnspr4.so.1 => /usr/local/lib/libnspr4.so.1 (0x28235000)
        libiconv.so.3 => /usr/local/lib/libiconv.so.3 (0x28265000)
```

Hence care should be taken to ensure one does not forget dependencies inadvertantly.

The program libchk starts from “standard” paths, /sbin, /sbin, /usr/bin /usr/sbin, /usr/libexec, and the same thing under /usr/local and /usr/X11R6. It then runs objdump -p on each binary and so discovers the needed libraries, without using “ldd”. For the same example it lists for example “NEEDED libnspr4.so.1”. It keeps all that in memory. It also runs ldconfig -r and keeps in memory the list of all libraries cached and their complete path. A typical output is “lssh.2 = /usr/local/lib/compat/libssh.so.2”. It then compares the two.

It also takes care of brandelf questions, Linux or FreeBSD binaries, etc., and it outputs a list of files without their corresponding libraries and libraries unassociated to binaries. This seems mainly useful to purge copies kept under /usr/local/lib/pkg, still if binaries depend on them and are not in standard location they will be screwed. Same thing for shared objects depending on such shared libraries like the above libmozjs.so.

6 The Debian package system.

The Debian system is like the FreeBSD system, a very strong set of rules and tools to produce packages. The tools are much more decentralized, there is no port tree in the FreeBSD sense. To produce a package one downloads the source code of software from the web, and one applies a set of tools and some editing work, which at the end produce a package, able to be kept in a Debian repository.

6.1 The side of source code.

Let us start describing the most user friendly way of doing that, using source packages preconfigured by Debian developers, and illustrate the case of our regular example atk.

The basic documentation to deal simply with source packages is [7] Chapter: Working with source packages. One also needs to install the dpkg-dev packet, and fakeroot, such as:

```
root# aptitude install dpkg-dev fakeroot
```

Then apt-get has shortcuts to download automatically the source code for a packet. First locate the correct name, using the apt cache sytem:

```
michel@tulipe:~$ apt-cache search atk1.0
libatk1.0-0 - The ATK accessibility toolkit
libatk1.0-dbg - The ATK libraries and debugging symbols
libatk1.0-dev - Development files for the ATK accessibility toolkit
libatk1.0-doc - Documentation files for the ATK toolkit
libatk1.0-data - Common files for the ATK accessibility toolkit
```

```
apt-get source libatk1.0-0
```

This downloads two files, the original atk sources and the Debian diffs, here the original tarball of source code, atk1.0_1.11.4.orig.tar.gz, and the set of Debian diffs, atk1.0_1.11.4-0ubuntu1.diff.gz, extracts and patches everything in a directory atk1.0-1.11.4. Then one goes to this directory and run:
`dpkg-buildpackage -rfakeroot -uc -b`
This does everything necessary and at the end one finds several Debian packages in the upper directory, such as libatk1.0-0_1.11.4-0ubuntu1_i386.deb which are directly installable with dpkg -i.

This works the following way: the Debian diff contains the whole content of a "debian" subdirectory, which keeps the necessary tools and metadata, plus appropriate patches. If we go to the directory that dpkg-buildpackage has created, the source code is patched, and there is a new directory "debian". This directory contains all sorts of files necessary for Debian magic, but the main one is the file "rules" which is an executable script. In fact the package build is launched by running this script. Here it is quite short:

```
#! /usr/bin/make -f

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/rules/simple-patchsys.mk
include /usr/share/cdbs/1/class/gnome.mk
include /usr/share/gnome-pkg-tools/1/rules/uploaders.mk

DEB_CONFIGURE_SCRIPT_ENV += LDFLAGS="-Wl,-O1 -Wl,-z,defs -Wl,--as-needed"

# Speed up loading.
LDFLAGS += -Wl,-O1

# build static stuff:
DEB_CONFIGURE_EXTRA_FLAGS := --enable-gtk-doc --enable-static

SHLIB_VERSION=1.9.0
DEB_DH_MAKESHLIBS_ARGS_libatk1.0-0 := -V"libatk1.0-0 (>= $(SHLIB_VERSION))"

DEB_DH_STRIP_ARGS := --keep-debug -Xlibatk1.0-udeb

# Yes, this is a hack, but dh_strip --dbg-package creates a libatk1.0-0-dbg
# and we don't want to change libatk1.0-dbg name for the moment.
binary-fixup/libatk1.0-dbg::
	mv debian/libatk1.0-0/usr/lib/debug/ debian/libatk1.0-dbg/usr/lib/
```

```
install/libatk1.0-udeb::
        cp debian/tmp/usr/lib/libatk-1.0.so.0 debian/libatk1.0-udeb/usr/lib/
```

The script immediately executes the "make" command, and we have something very similar to what happens under FreeBSD. At the end, the "debian" directory contains a subdirectory libatk1.0-0, in which all the build products can be found, and only them. Going to this directory one finds two more directories: "DEBIAN" which contains the metadata for the package system, `control md5sums postinst postrm shlibs`, and "usr" which has "lib" and "share", where lib contains the libatk library, and "share" its documentation. To obtain the .deb package it is only necessary to do something like a compressed tarfile of this stuff. Indeed, running "dpkg-deb -e" on the .deb produced above one gets the content of the DEBIAN directory, and running "dpkg-deb -x" one gets "usr" and everything below.

For our purposes it is therefore interesting to look at the contents of DEBIAN, which is the specific Debian metadata in a package. First the content of "control", something very similar to the pkg-descr of FreeBSD, but with added dependency information, as in +CONTENTS. Note the particular form of the dependencies with order operators, that we shall comment later on. There are also specific Debian categories, such as recommended packages to install.

```
Package: libatk1.0-0
Version: 1.11.4-0ubuntu1
Section: libs
Priority: optional
Architecture: i386
Depends: libc6 (>= 2.3.4-1), libglib2.0-0 (>= 2.10.0)
Recommends: libatk1.0-data
Installed-Size: 156
Maintainer: Akira TAGOH <tagoh@debian.org>
Source: atk1.0
Description: The ATK accessibility toolkit
 ATK is a toolkit providing accessibility interfaces for applications or
 other toolkits. By implementing these interfaces, those other toolkits or
 applications can be used with tools such as screen readers, magnifiers, and
 other alternative input devices.
 .
 This is the runtime part of ATK, needed to run applications built with it.
```

The information here is generated by using the file "control" which is located in the "debian" directory of the source code, and is hand edited by the developer. As an example, if we look at the source package "apt" which provides apt-get and other binary packages, the "control" file is:

```
Source: apt
Section: admin
Priority: important
Maintainer: APT Development Team <deity@lists.debian.org>
Uploaders: Jason Gunthorpe <jgg@debian.org>, Adam Heath <doogie@debian.org>,
Matt Zimmerman <mdz@debian.org>, Michael Vogt <mvo@debian.org>
Standards-Version: 3.6.1
Build-Depends: debhelper (>= 4.1.62), libdb4.3-dev, gettext (>= 0.12)
Build-Depends-Indep: debiandoc-sgml, docbook-utils (>= 0.6.12-1)

Package: apt
Architecture: any
Depends: ${shlibs:Depends}
Priority: important
Replaces: libapt-pkg-doc (<< 0.3.7), libapt-pkg-dev (<< 0.3.7)
Provides: ${libapt-pkg:provides}
Recommends: ubuntu-keyring
Suggests: aptitude | synaptic | gnome-apt | wajig, dpkg-dev, apt-doc, bzip2,
gnupg
Section: admin
Description: Advanced front-end for dpkg
 This is Debian's next generation front-end for the dpkg package manager.
 It provides the apt-get utility and APT dselect method that provides a
 simpler, safer way to install and upgrade packages.
 .
 APT features complete installation ordering, multiple source capability
 and several other unique features, see the Users Guide in apt-doc.

Package: apt-doc
Architecture: all
Priority: optional
Replaces: apt (<< 0.5.4.9)
Section: doc
Description: Documentation for APT
 This package contains the user guide and offline guide, for APT, an
 Advanced Package Tool.
.....
```

As we can see, for each of the binary packages generated from source, the required information is specified here. This is also the occasion to show the other categories Debian introduces, besides pure dependency: Recommends and Suggests.

The file md5sum contains hashes of the shared library and of the documentation. The files postinst and postrm are very important files related to the elaborate installation procedures for a package that Debian enforces. Here postinst is executed after installation of the shared library, and postrm after its removal. Not surprisingly:

```
postinst
#!/bin/sh
set -e
# Automatically added by dh_makeshlibs
if [ "$1" = "configure" ]; then
        ldconfig
fi
# End automatically added section
postrm
#!/bin/sh
set -e
# Automatically added by dh_makeshlibs
if [ "$1" = "remove" ]; then
        ldconfig
fi
# End automatically added section
```

that is instructions to run ldconfig to sync the cache. There may be preinst scripts and things of that sort.

This is more formal that what can be found with FreeBSD, but functionally not different from the instructions found in the +CONTENTS file of a package.

Finally there is a file "shlibs" for which i don't know FreeBSD equivalent, which contains only:
`libatk-1.0 0 libatk1.0-0 (>= 1.9.0)`
The signification of this information is given in the Debian policy manual, the equivalent of the Porter's handbook [8] in 8.6.3 The shlibs File Format, so libatk-1.0 is the library name, 0 is the soname version number, and the rest indicate dependencies, that is "which packages are required to satisfy a binary build against this library". So we need the package libatk1.0-0 with a sufficiently high version number.

This is characteristic of Debian system, and i think has more flexibility than the FreeBSD system. Quoting the policy manual for an example:

```
Package: mutt
     Version: 1.3.17-1
     Depends: libc6 (>= 2.2.1), exim | mail-transport-agent
```

We see here something related to version 1.3.17-1 of the mutt package. It requires at least version 2.2.1 of libc6 package, but any superior version will do. Incidentally we have also dependence on a mail transfer agent which may be either exim, or an abstract target. The abstract target will be fulfilled by sendmail or postfix.

To summarize, the structure of a Debian .deb package is astoundingly similar to the structure of a FreeBSD package. One can see difference in some details such as the shlibs file and the elaborate notation for "relationship fields". Such notation with relational operators has been introduced in the pkgsrc NetBSD system.

To complete this brief survey, let us talk a little about the tools which allow to build more easily the infrastructure in the "debian" directory. One needs to install the package dh-make which contains the command dh_make. Since this works particularly well for GNU software which has "configure" files, i download and extract bison-2.3, and descend in its directory, then type dh_make:

```
michel@tulipe:~/bison-2.3$ dh_make

Type of package: single binary, multiple binary, library, kernel module or
cdbs?
 [s/m/l/k/b] s

Maintainer name : Michel Talon
Email-Address   : michel@
Date            : Sat,  2 Sep 2006 16:25:14 +0200
Package Name    : bison
Version         : 2.3
License         : blank
Type of Package : Single
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. bison
uses a configure script, so you probably don't have to edit the Makefiles.
```

This creates a debian subdirectory with the infrastructure, and keeps a copy of the original bison source in ../bison-2.3.orig. In particular an executable script debian/rules, which starts with `#!/usr/bin/make -f`. We can edit stuff here. There are several targets, in particular build.

```
michel@tulipe:~/bison-2.3$ debian/rules build
dh_testdir
# Add here commands to configure the package.
...
The configure script runs
...
The compilation runs
```

```
...
#docbook-to-man debian/bison.sgml > bison.1
touch build-stamp

michel@tulipe:~/bison-2.3$ debian/rules install
dh_testdir
dh_testroot
dh_testroot: You must run this as root (or use fakeroot).
make: *** [install] Erreur 1
michel@tulipe:~/bison-2.3$ fakeroot debian/rules install
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs
# Add here commands to install the package into debian/bison.
/usr/bin/make install DESTDIR=/home/michel/bison-2.3/debian/bison
make[1]: entering directory  /home/michel/bison-2.3
Making install in build-aux
...

michel@tulipe:~/bison-2.3$ ls -R debian/bison
debian/bison:
usr

debian/bison/usr:
bin  lib  sbin  share

debian/bison/usr/bin:
bison  yacc
...

michel@tulipe:~/bison-2.3$ fakeroot debian/rules binary
...
a bunch of make commands
...
dh_testdir
dh_testroot
dh_installchangelogs ChangeLog
dh_installdocs
dh_installexamples
dh_installman
dh_link
dh_strip
dh_compress
dh_fixperms
```

```
dh_installdeb
dh_shlibdeps
dh_gencontrol
dpkg-gencontrol: warning: unknown substitution variable ${misc:Depends}
dh_md5sums
dh_builddeb
dpkg-deb: making package bison in ../bison_2.3-1_i386.deb.
```

We see how the system creates a fake installation directory in debian/bison, and how it needs faking to be root to do that. This is the aim of the command fakeroot which preloads a library modifying some calls such as geteuid(), chown(), etc. As the consequence one has the illusion of being root, while one can only modify the files one owns, of course.

At the install stage, bison has been "installed" in this fake installation directory, but not Debian metadata. Anyways this solves the problem of the non automatic filling of the packing list which plagues FreeBSD. Here the complete files of the packages are isolated in debian/bison and are used in the last step "debian/rules build" to build the package. This uses several scripts as we can see. Now we have

```
michel@tulipe:~/bison-2.3$ ls debian/bison/DEBIAN/
control  md5sums
```

and the .deb package exists and is functional. If i install it with dpkg -i it will be registered in the system and will give a perfectly functional bison.

All this without a single manual intervention. Of course, this package will not be fully conformant to Debian policy, so some manual polishing of debian/rules and other files in the debian directory should be necessary to get an acceptable package for public use. This type of work is what "Debian developers" are doing. Software which doesn't have configure scripts usually require much more manual intervention, although dh_make will always produce the basic infrastructure, but having less clues, some informations will be missing.

6.2 The side of packages.

The Debian system works by providing repositories with precompiled packages. One can also get the corresponding source packages, as shown above. These repositories are mirrorred, and of course are scanned by the apt system to know the available packages. What turns a directory containing a bunch of .dev packages into a repository is simply the presence of a file Packages.gz which lists all packages in the directory. This is the file the

apt system reads to know what is available here. It is something like the INDEX file of the FreeBSD system. To create this file in the directory, one runs:
`dpkg-scanpackages directory | gzip > directory/Packages.gz`
In fact dpkg-scanpackages scans each .deb to obtain metadata information and it is dispalyed in Packages.gz in the following form:

```
Package: libatk1.0-0
Priority: optional
Section: libs
Installed-Size: 188
Maintainer: Akira TAGOH <tagoh@debian.org>
Architecture: i386
Source: atk1.0
Version: 1.11.4-0ubuntu1
Depends: libc6 (>= 2.3.4-1), libglib2.0-0 (>= 2.9.3)
Recommends: libatk1.0-data
Filename: pool/main/a/atk1.0/libatk1.0-0_1.11.4-0ubuntu1_i386.deb
Size: 71138
MD5sum: daabcca7cd9fa5bb9379315da73efe65
Description: The ATK accessibility toolkit
 ATK is a toolkit providing accessibility interfaces for applications or
 other toolkits. By implementing these interfaces, those other toolkits or
 applications can be used with tools such as screen readers, magnifiers, and
 other alternative input devices.
 .
 This is the runtime part of ATK, needed to run applications built with it.
Bugs: mailto:ubuntu-users@lists.ubuntu.com
Origin: Ubuntu
Task: ubuntu-desktop, edubuntu-desktop, xubuntu-desktop
```

After having read that, apt caches all this information locally in a database. It can be queried using "apt-cache". It is remarkable that all these operations, running dpkg-scanpackages, downloading and parsing it, answering queries, etc. is always fast.

6.3 Apt.

We have seen above example of the use of apt-get to either install packages or download their source code. Similarly it can be used to completely upgrade a machine. Running "apt-get update" updates the cache of the Debian repositories contents. Running "apt-get dist-upgrade" upgrades the entire machine including base system and kernel. I have juste upgraded my Ubuntu installation due to regular maintenance. Around 130 packages were downloaded and installed, including the X server, several Gnome components, new kernel, etc. while i was still working under Gnome typing the above text. This took no more than a quarter of an hour. A reboot after, everything worked perfectly OK with new kernel, Grub had been automatically edited to load this new kernel, etc. Compared with what portupgrade has to offer, the apt efficiency is phenomenal.

This is probably because apt has been entirely coded in C++, with great care taken for efficiency of operation, efficient caching of state, etc. This means a large quantity of code, very clearly written, taking care of all aspects of the problematic of package management. A particular interesting feature of apt, from our point of vue, is that apt has been documented by the developers for developers. This documentation explains many of the problems which can occur in a package management system, and the solutions they have devised. As such it is of general applicability. To obtain it, one installs the package `libapt-pkg-doc` on a Debian machine, or simply get it here [9].

More recently has been introduced an enhancement, called "aptitude" which basically does the same thing as apt-get, and moreover keeps track of the packages that have been installed as dependencies of a package you have been required. If later on you decide to remove this package it will also remove all those dependencies provided they are not used by some other live package. This helps keeping the computer uncluttered. Aptitude also has a curses interface, similar to that of deselect, that is particularly user unfriendly. But there are easy to use graphical front-ends running under X, like synaptic, or even simpler the "add/remove" widget of Ubuntu. To summarize, the apt system allows Debian based distributions to have a fantastic install and upgrade management system. It simply works, and is fast.

7 Conclusion.

We have described two systems supposed to allow access to a large number of software packages, and are generally considered to be widely different. FreeBSD is mainly source based, where Debian is mainly binary based, at least in the perception of people. In fact we have seen that in the step going from source to package, both systems are very much alike, and the structure of a Debian package is not much different from that of a FreeBSD package. Similarly the solutions for package management, that is how to install and upgrade a large number of interdependent packages use essentially the same ideas, be it portupgrade for FreeBSD or Apt for Debian. However the efficiency is radically different, apt works ways better than portupgrade, at least in terms of speed. But the main factor ensuring reproducibility and reliability of the apt system is working with binary packages. You can be sure at least of the existence of a binary package, and probably that it works, due to the severe testing in the Debian system. There is no guarantee in a source based system. Hence no package management system can be reliable, however sophisticated it is. A possible cure to this problem, is to build all required packages on a test machine, or in a jail. That will obviously solve the main problem. But this puts considerable burden on the end user. At the moment, there are efforts amongst FreeBSD developers to better manage some of these problems. For example, the port devel/portmk is aimed at rewriting bsd.port.mk, the port Tool/portbuild has all sorts of scripts to allow package building in a chroot setup.

References

[1] `http://www.netbsd.org/Documentation/software/pkg-wildcards.html`

[2] `http://www.netbsd.org/Documentation/software/pkgviews.pdf`

[3] `http://www.netbsd.org/Documentation/pkgsrc/`

[4] `http://sources.gentoo.org/viewcvs.py/portage/main/trunk/pym /portage.py?rev=4374&view=txt`

[5] `http://www.freebsd.org/doc/en_US.ISO8859-1/books/porters-handbook/`

[6] `http://www.lpthe.jussieu.fr/~talon/portindex-18_1.tgz`

[7] `http://www.debian.org/doc/manuals/apt-howto`

[8] `http://www.debian.org/doc/debian-policy`

[9] `http://www.lpthe.jussieu.fr/~talon/aptdoc.tgz`

[10] `http://www.lpthe.jussieu.fr/~talon/pkg_check.py`

[11] `http://www.lpthe.jussieu.fr/~talon/pkgupgrade`

[12] `http://www.lpthe.jussieu.fr/~talon/pkg_save.py`

[13] `http://www.lpthe.jussieu.fr/~talon/show_index.py`