NetBSD/Desktop: Scalable Workstation Solutions

Jan Schaumann – Stevens Institute of Technology

Abstract

As a mature operating system with emphasis on code quality and standards compliance as well as an abundance of third-party applications readily available, NetBSD offers an easily deployable and user-friendly desktop solution. Managing large numbers of identical workstations can be facilitated through the use of a dedicated build server and an IPSec'd server-push strategy to update the clients; a strategy which has proven to be reliable, secure and scalable.

1 Introduction

While BSD in general and NetBSD in particular has had a reputation for being primarily a reliable and secure operating system for servers, it is (in the mainstream, at least) still not considered "user-friendly enough" for the desktop. Yet aside from the sheer number of available applications there are distinct advantages to choosing NetBSD, a complete open source operating system with emphasis on code quality and standards compliance, *especially* when it comes to managing a large number of identical desktop workstations.

This paper will elaborate on these advantages as well as present techniques and strategies to install, maintain and update large installations, accounting for the various needs of many hundreds of users and the implied complexities of thousands of third-party applications.

The infrastructure presented has been in production use in the Department of Computer Science at Stevens Institute of Technology for over 3 years, where it is used to manage several public laboratories as well as most of the faculty workstations. It consists of a dedicated build server and an IPSec'd server-push strategy to update the clients, providing scalable, secure and easy updates of the operating system as well as of all third-party application.

2 Why NetBSD?

In order to see why NetBSD would be a good solution for a regular desktop system, we should investigate what exactly is required of such a workstation and how it can be easily duplicated and deployed in a large environment. In order to address the first point, we need to look at the needs of the target users:

Not very surprisingly, given the obvious heritage of UNIX and BSD, Unix-like Operating Systems (OS) have long been the preferred choice in academic institutions in general and the field of Computer Science in particular. All users in such an environment (faculty, researchers, students) have high expectations of the robustness of the OS and its performance. At the same time, the widely different research interests require a large number of at times specialized applications to be made available to them.

2.1 What does "user-friendly" mean, anyway?

Many people expect a desktop OS to be "user-friendly", without clearly understanding the meaning of this phrase. As should become immediately clear from the varying needs and expectations of a multifaceted user-base, it can mean different things to different people, based on their background knowledge, experience and interest. It would be prudent to characterize any OS that allows these users to get their work done efficiently to be "user-friendly".

Oddly enough, the phrase "user-friendly" is often quoted in the context of OS installation, software installation and upgrades, security patches, and general maintenance. The procedures involved in these tasks are not considered *user*-friendly. This reveals the common misconception that a

"desktop" or a "workstation" is a single, autonomous machine used (and maintained!) by a single user.

Actually, the tasks most often described as not being "user-friendly" are not – and *should* not be – performed by the *user*, but rather by the administrator. Maintaining the complex dependency-tree of installed applications, tweaking the OS to remain performant under duress and keeping dozens or hundreds of workstations in sync as well as available to all users are not tasks for a regular user and pose an entirely different set of requirements to the OS.

In the end, it boils down to this: The user of the desktop must get work done. The administrator must be able to maintain such desktop systems easily and efficiently. What we need is an "*admin*-friendly" OS with "*user*-friendly" applications!

2.2 Requirement: User-friendly

From the user's point of view, all interactions with the OS are characterized by the applications available on this platform. As long as the right tools for the job are available, the user may remain oblivious as to what exactly goes on "behind the curtain".

While commercial vendors release their software for only a limited number of unix-like OS, the vast majority of the available Open Source Software (OSS) – ranging from small command-line tools to entire desktop environments or office suites – is written for one or another version of Unix and can (often easily) be ported to another flavor.

In a multi-OS environment, it is beneficial to the user if she can use the same general desktop software and tools on one host as on another. Maintaining the same set of installed applications at the same version across even different OS helps avoid confusion when it comes to the user-interface or a user's configuration files.

Another important and often neglected aspect of user-friendliness is the combination of, on the one hand, hiding unnecessary complexities from one set of users while, at the same time, catering to the expertise of another set:

Since the target user-base includes people with very specific interests in the field of networking, operating system design, and system security, as well as students who learn to understand these concepts and develop software, the OS provided to them should allow them to use applications that are not traditionally part of a so-called desktop system. This includes access to the software development tools such as different compiler toolchains and various scripting languages, as well as the more typical desktop applications. In addition, it is often beneficial to provide the full sources for the OS in use to allow these users to look "under the hood" and understand how the OS, and the tools provided by it, work.

2.3 Requirement: Admin-friendly

As explained in section 2.1, the user-friendliness of an OS and its applications is necessary, but not *sufficient* to make a sensible choice for your environment. We also need to take a look at the qualities of an OS from the other point of view. In fact, given that the majority of the software in use is available for most unix-like OS, it seems that the admin-friendliness of an OS might be the more important aspect of all!

A suitable OS should be easy to install, maintain and update. A dependency on a specific hardware or software vendor is equally undesirable as too inconsistent a base system. On the one hand, we would want a *complete* OS with all base applications (ie the *userland*) from the same source tree; this allows for easy system maintenance and OS upgrades. At the same time it's important to ensure that all required software applications are available; finally, we need to anticipate future requirements.

With respect to hardware requirements, we must be able to support state-of-the-art equipment without having to resort to "bleeding edge" development snapshots. This, together with the need for the necessary security patches, requires an OS with a regular and reliable release engineering cycle. A mature and well thought-out third-party application framework or package management system is equally important since the vastly different needs of the many users imply a complex set of software dependencies. Finally, we must be able to run a few commercial applications that may not be available for all OS.

2.4 So... why NetBSD?

Looking at the requirements and the target user base as analyzed and described above, we see that NetBSD is a near perfect match for this environment.[1] NetBSD has a strong following among system administrators: it provides a *complete* OS, easily maintained in a single source tree and has an outstanding security track record. No need to try to track down and follow different development branches for each and every single userland binary, nor is it necessary to adhere to one commercial vendor's idea of release engineering and hope that other software vendors will adapt their packages accordingly.

Through the use of binary emulation it is possible to run a large number of commercial applications which may only have been released for another OS.[2]

[1] This is not to say that other OS choices would not be justifi able; however, the intimate familiarity with and knowledge of NetBSD by the administrators clearly make it the easier choice. It is also worth noting that over the years, experience has taught us that other, more popular choices would likely have caused us more headaches.

[2] At Stevens, the most important applications in this category are Sun's JDK's, MathWorks' MATLAB and Maplesoft's Maple; all industry standard applications which perform without performance penalty using NetBSD's Linux emulation layer.[3]

Finally, installation of third-party software and maintenance of all installed software is simplified by using the NetBSD Packages Collection[2] (aka *pkgsrc*), a source based software package management system. Through consistent use of pkgsrc and its cross-platform features, not only are we able to track over 1000 different applications and their interdependencies, but we are also able to provide the same number of applications in the same versions across different OS such as IRIX or Linux when this is necessary.

All currently used hardware – from regular single-processor desktop workstations to multi-processor, SATA-RAID enabled servers with support for fibre-channel or optical gigabit network devices – is fully supported by NetBSD. Its conservative release cycle allows us to rely on the hardware being fully functional and well-tested rather than having to make use of development features in unstable drivers. At the same time, the progressive nature of NetBSD in areas such as support for AMD's 64bit processors or IPv6 networking and its focus on standards compliance ensures that we will be able to add new hardware as it becomes available while at the same time continuing to provide our researchers with the most suitable reference platform available.

3 Infrastructure

At Stevens Institute of Technology, NetBSD is now used throughout the Department of Computer Science and the Department of Mathematics to maintain the Unix desktops and public laboratories in addition to a number of administrative machines.[3]

In order to maintain these machines, we have developed a set of administrative scripts that make up an infrastructure consisting of a dedicated build server and an IPSec'd server-push strategy to update the clients, providing scalable, secure and easy updates of the OS as well as of all third-party application. This setup, which has been in production use for over three years, reduces the administrative overhead involved in managing a large number of virtually identical hosts immensely and makes a trivial task integrating new machines into the framework.

All workstations have similar hardware, which allows us to run a common yet tailored kernel. The kernel contains the drivers for all available hardware throughout the system, thus allowing us to replace, for example, a network or a graphics card without having to recompile the kernel, while at the same time not suffering the performance penalty of a GENERIC kernel that includes support for a multitude of devices that are not required.

[3]The Department of Computer Science also maintains a state-of-the-art clustered High Performance Computing Facility suitable for research in areas of computer science and engineering that may require substantial computational effort.[4] A second, similar cluster within the Department of Physics is currently being converted to NetBSD as well.

Since all workstations contain an identical software image, we can easily minimize downtime of a single system: if a machine goes down due to a hardware failure, we can quickly and easily replace the failed component, bring the machine back up and thus allow the user to continue to use the resources while we troubleshoot the failed device.

Statistical details about the infrastructure presented here can be found in Figure 1; the general setup is described in Section 3.1.

# of administrative scripts	7
total LOC of administrative scripts	388
# of users	approx. 2900
# of workstations	70
# of third-party packages not under pkgsrc	7
# of third-party packages under pkgsrc	1054
size of workstation image	9.3 GB

Figure 1: The system in numbers

3.1 Server Configuration

The build server, known by its hostname, *amstel*, used to maintain the workstation image, is a dual-processor i386 machine with 2 GB of RAM, powerful enough to build all required binary packages and the NetBSD kernel and userland in an acceptable timeframe. It has enough diskspace in a RAID 5 disk array to host the pkgsrc and regular src trees used, the workstation image (itself described in more detail in section 3.2) and provides enough temporary scratch space for the build process.

The machine hosts a total of three pkgsrc trees: one for the latest stable branch of pkgsrc (I), one for the HEAD of pkgsrc (II) and a third tree which incorporates parts of the HEAD with the latest stable branch as well as some local modifications (III). The first two are updated regularly from anonymous CVS sources and are used as a reference for the third one, which is itself used to build packages for the workstation. Trees (I) and (II) are also exported via NFS through a local gigabit network to the other non-NetBSD servers that use pkgsrc. Tree (III) is mounted via a null-mount from within two specific chroots.

The first chroot on the server is located in `/new` and represents the workstation image in use on all clients. The second is located in `/sandbox`, which represents the "playpen" for the workstation image. That is, the workstation image is duplicated in this directory and all software updates or additions are performed in this directory first. Packages are built and installed here, then turned into binary packages which can then be added in `/new` using the `pkg_*` tools. This setup makes it possible to update the userland using the standard NetBSD build.sh build mechanism (see "Using the build.sh Front End" in [5]) by simply defining `DESTDIR=/sandbox` in `/etc/mk.conf`.

In general, `/new` is not modified directly and only binary packages are handled in this chroot, but every now and then it is necessary to perform one of the various package-related administrative activities in the `/new` chroot (after carefully testing it in `/sandbox`, of course), which is why the second null-mount is necessary.

A nightly cronjob checking the set of installed packages (in each of the server's base, the sandbox and the production workstation image) against a list of known vulnerabilities[4] ensures that we are alerted of any security issues as soon as a vulnerability is known.

3.2 Client Configuration

The disk image of the workstations is, as mentioned previously, stored in `amstel:/new`. It consists of a standard NetBSD/i386 installation together with a large number of third-party applications in order to cater to the different needs of the different users. Among those applications are most common window managers and desktop environments (including WindowMaker, KDE and GNOME), various browsers (including Netscape, Firefox, Mozilla and Opera), numerous software development tools (ranging from autoconf and automake to full-featured IDEs like eclipse), and programming languages (such as Sun's JDKs, Python, Perl, Ruby and Scheme), specialized commercial applications (including MathWorks' MATLAB and Maplesoft's Maple) and office suites and applications (including OpenOffice, gnumeric, gnucash, abiword and KOffice). In other words, it includes a complex and full desktop environment for a multitude of different users.

In order to allow us to swap parts easily among all the different workstations or even replace an entire machine with a minimum downtime for the user, all workstations are kept identical with respect to the software installed. Since some of the hardware differs (most notably graphics and network cards), there are a few differences in a limited number of configuration files, which therefore need to be kept exclusive to each host in addition to the obviously security-related files:

```
etc/X11/XF86Config
etc/master.passwd
etc/racoon/psk.txt
etc/rc.conf
etc/spwd.db
etc/ssh/ssh_host_dsa_key.admin
etc/ssh/ssh_host_dsa_key.admin.pub
etc/ssh/ssh_host_key.admin
etc/ssh/ssh_host_key.admin.pub
etc/ssh/ssh_host_rsa_key.admin
etc/ssh/ssh_host_rsa_key.admin.pub
etc/printcap
```

[4] using pkgsrc's *audit-packages*

The list of files is rather self-explanatory, though it might be worth pointing out that all machines share the same default ssh-configuration and -keys. This is due to the fact that the hostname "lab.cs.stevens-tech.edu" is a round-robin of a number of these workstations, allowing users to connect to a generic name, yet spread the load across multiple machines.[5] Using a common key for all machines, however, necessitates running a second ssh dæmon on an alternate port for administrative purposes, which explains the distinct files in `etc/ssh`.

In order to keep these files separate from the rest of the workstation image, we created one directory for each client's `etc` directory in `<hostname>-etc` as well as one directory `amstel:/new/etc` which contains all other files usually found under `/etc` on a NetBSD host. Section 5.1 explains how the script used to update a workstation processes these directories.

4 Software installation

All software installed on the workstations that is not part of the NetBSD base system is installed, if possible, from the NetBSD Packages Collection (see [1] and [2] for detailed documentation). Any required piece of software that is not part of pkgsrc must be carefully investigated: if it is OSS or otherwise publicly available, we create the appropriate package and feed the changes back into the NetBSD pkgsrc CVS tree. If it is not suitable for inclusion in the Packages Collection, then the software is installed in `amstel:/new/usr/local`. As can be seen from Figure 1, we are fortunate enough to only have a very small number of applications that cannot be controlled through pkgsrc; these applications include mostly commercial, specially licensed software and a number of homegrown applications and scripts.

The pkgsrc trees in use are updated via CVS from the local anoncvs mirror on a regular basis. Due to the large number of applications installed and the resulting, rather complex dependency-tree we are taking a more conservative approach with the production use pkgsrc tree. This tree (III in Section 3.1) is carefully merged and updated on an as-needed basis only. That is, if a newer version of a piece of software is required, it is first updated to the latest stable branch of pkgsrc. If any security fixes have not yet been pulled up to the stable tree or a newer version is required, the tree is updated to the HEAD or patches backported. A small number of packages have some local modifications applied as well.

The build process itself is done in the

[5] Of course this would also be possible with distinct ssh-keys, but we have found that especially among novice users and windows clients, multi-homed hosts with different ssh-keys often cause some confusion and generate warnings about the keys not matching the hostname. Rather than encouraging users to ignore such warnings, we decided to distribute a single key to all machines.

`amstel:/sandbox` chroot to ensure that the package (and all its dependencies and/or other packages that had to be updated as a result) can successfully be built, installed, cleanly deinstalled, reinstalled and tested. Binary packages are then created from within that chroot and the production use image is updated accordingly.

5 Update procedure

Updating all the workstations from the build server is done by means of an IPSec'd server-push performing several `rsync`-passes on the remote side (see Section 5.1 for details). The administrative scripts used to initiate the push allow for synchronization of all workstations or certain subsets based on department or laboratory in parallel or pushing just a single machines.

If the changes require additional testing, individual machines can be synchronized with the sandbox to allow for the installation or upgrading of large parts of the workstation image without risking breaking all clients in production use. In these cases a note explaining the currently tested update is placed into a special file (called "`dont`"; see Appendix A.2), the existence of which prevents a full push to all clients from taking place – a security precaution that has often proved useful.

Similarly, the setup also allows for synchronizing only parts of the filesystem: each pass can be performed individually. If, for example, a general system upgrade is still being tested on some individual workstations but an important update of one of the applications in `/usr/local` is pending, then only the update of that part of the filesystem can be pushed out to all clients, retaining the full update of the base system until it has been sufficiently tested.

If parts of the filesystem need to be (temporarily) excluded from being synchronized, they can be specified in specific configuration files as regular `rsync(1)` exclude patterns.

A push-strategy rather than a pull-strategy was chosen to prevent accidental deployment of not fully tested updates. Since a push is proactive, it forces the administrators to think about what changes are ready for deployment and carefully evaluate when a complete push of all workstations should take place. A client-pull mechanism might easily be forgotten and cause serious downtime if any software updates were accidentally left incomplete (either as a result of human negligence or of a software failure during the upgrade process). Also, as can be seen from the above, a server-push allows for much finer control, as a client-pull would have to be automated and thus complete.

5.1 The push script

The shell script `push.sh` (the full script can be found in Appendix A.1) is used to update a single workstation after changes have been made to the workstation image. It uses `rsync(1)` to update the remote host (see section 5.2 for security considerations). To summarize, `push.sh` performs the following steps:

1. **Enter new information in `/new/etc/updates`.**
   All changes to the workstation image are briefly noted in the file `/usr/local/stevens/UPDATES`. This file helps us keep track of what changes were made and, more importantly, why they were made. The `push.sh` script timestamps this file and places a copy into `/new/etc/updates`, so that users can always review the latest changes and the administrator can easily determine if any given workstation is currently up-to-date.

   This file is also (manually) emailed in a PGP-signed message to a local mailing list. This, too, allows users to keep up to date with changes on the system while at the same time providing some log of the changes.

2. **Set up exclusions.**
   It may be desirable or necessary to exclude certain files on a specific host from being synchronized with the rest. Any such files can be entered in a separate file which is parsed by `push.sh` in order to create a list of exclusions.

   A reason for such an exclusion might be a machine that requires a different kernel from all the others, but that is otherwise identical to the normal workstations.

   Note that these exclusions are set up on a per-host basis. If general site-wide exclusions need to be set up, they can be entered in a different file using standard `rsync(1)` exclude patterns.

3. **Run any remote commands if necessary.**
   Depending on the changes pushed out, it may be necessary to first run a specific command on the remote host, for example to stop a service or to unmount a partition. If `push.sh` finds the file `beforesync` in the local directory, it is copied to the remote host where it is executed as a shell script.

4. **Do passes as desired.** The actual push process is divided into several *passes* which may be specified individually. If no pass is explicitly specified, all default passes are performed consecutively. This allows for fine-grained control over which parts of the filesystem are updated and is necessary to set up the appropriate exclusions.

   Each pass inspects a special file containing these exclusions as a list of pathnames (directories or files) which should be ignored when running `rsync(1)`.

5. **Do absolute copies if necessary.** Similar to the exclusion set up in step two, this step allows special files to be

copied to a specific absolute destination on the remote host.

6. **Run any remote commands if necessary.** Depending on the changes pushed out, it may be necessary to run a specific command on the remote host after all changes have been pushed out; for example, to restart a service after the configuration file was changed.

   If `push.sh` finds the file `aftersync` in the local directory, it is copied to the remote host where it is executed as a shell script.

As mentioned above, this script only updates a single host. A number of other scripts are used to update all available workstations or a subset thereof (see Appendix A.2). In combination with these scripts, `push.sh` provides a sufficient amount of flexibility for most situations. For example, if a major package build is still in progress (for example, KDE needs to be updated) but a new release of one of the commercial applications installed is available, it would be trivial to push out the `/usr/local` hierarchy to all clients, while retaining `/usr/pkg` for another time to allow for more testing.

On the other hand, it may be desirable to simply run a specific command on all hosts or to update a single file. The script would be able to achieve this, but it would involve some performance overhead. Therefore we have added a few simple scripts to perform just these tasks (Appendices A.3, A.4).

5.2 Security considerations

Since the entire filesystem for each client is transferred over the network, there are a number of security aspects to be considered. First, we need to ensure that only machines that are known are allowed to synchronize with the server. Second, we need to make sure that sensitive files are not transferred in the clear.

As explained in Section 5, we chose a server-push strategy, which partly addresses the first problem: no client can initiate the update, so that it's not possible for a Trojan to connect to the network, steal a known IP address and request an update. However, it would still be possible for an adversary to pose as a normal workstation and wait for the update to be pushed out. Fortunately, that, too, is not possible, as the only two ways we allow the `rsync(1)` processes to perform is either tunneled through `ssh(1)` or by use of `rsh(1)` over mandatory IPsec.

Both of these approaches require authentication, either by use of the private ssh-key for the dæmon listening on an alternate port or by use of IKE, IPSec'd dynamic encryption key exchange protocol. In our setup, we use `racoon(8)` with a pre-shared secret key stored in `/etc/racoon/psk.txt`. The permissions on this file – just like on the private ssh keys in `/etc/ssh/` – do not allow regular users to read these, so it should be impossible for the adversary to pose as the real client when a push is initiated by the server. Similarly, we have a specific ssh-key set in `/root/.ssh/authorized_keys` (the private key for which is only on the non-publicly accessible build server) that is used for synchronizing the machines if ssh is used as the remote shell, allowing access by this key only from the build server.

type	remote shell	time
minor updateem +	rsh + ipsec	332.27s → 5.5m
minor update	ssh	459.48s → 7.6m
major update*	rsh + ipsec	474.42s → 7.9m
major update	ssh	494.91s → 8.25m
full update#	rsh + ipsec	1307.71s → 21.8m
full update	ssh	1426.22s → 23.8m

+ update of a few files or a small package
* update of at least three large packages (such as mozilla, KDE etc.)
# update of the entire userland and several large packages

Figure 2: Scalability of a single push

It might be possible to run nearly the entire push unencrypted using regular `rsh(1)` for `rsync`, but clearly there are a few files that must not be transmitted in the clear (`/etc/master.passwd` and the files from the previous paragraph, for instance). However, if this approach were pursued, then the push script would need to switch the mechanism used to log in the remote machine based on the files to be transferred. This does not prove to be scalable, especially since packages are added that might also require encryption during file transfer (for example the `sudo(8)` package). In addition, we would loose the authentication mechanism provided by IPsec, which is why in the end we decided to make use of IPsec for all `rsh`/`rlogin` processes mandatory. Once we enabled IPsec, we quickly realized that another beneficial side-effect was that it allowed us to let `syslogd(8)` log all events to *amstel* in a secure fashion as well.

5.3 Scalability

When expanding the network of clients under control of this system, it is important to consider how well the system scales, how long each process takes, and what the penalties are of choosing one approach over another. The main factors with respect to scalability are:

- the time it takes for a single complete update
- the time it takes for all clients to be updated

The time it takes for a complete update is influenced by the number of changes since the last update, as well as which push-strategy (ssh or rsync over IPsec) is chosen. In production use there may often be miniscule changes (i.e. changes

of only a few files) just as there may be rather large updates (such as a rebuild of a significant portion of the binary packages) that need to be pushed out. Figure 2 shows the time in seconds for a few example changes including the worst-case scenario: an update of the entire base system as well as a number of large third-party updates.

Another factor in the calculation of these numbers is the amount of RAM available on the client. The majority of the workstations have 512 MB RAM, but a number of them still have only 256, while a few have as much as 1 GB RAM. The numbers in Figure 2 and Figure 3 are based on the average of repeated pushes of machines with 256 MB RAM, with the highest and lowest results being dropped.

The time it takes for all clients to be updated obviously depends on the time for each individual client, but also on how many hosts can be pushed out in parallel without saturating the network connection or overloading the server. Figure 3 shows the results of updating several sets of machines.

6 New workstation installation

Adding a new workstation into this setup is trivial. After the hostname of the new machine and its intended physical location (which determines the subnet the new host will be on) has been specified, we generate a new configuration for this client by using the script `gen-conf.sh` (see Appendix A.5). To complete the setup on the server side, all we need to do is add the new hostname to the appropriate collection of machines that allow us to push subsets based on location.

The actual installation process is initiated by booting off a NetBSD install floppy or CD-ROM. Instead of performing the regular installation that NetBSD's `sysinst` tool would usually initiate, we abort the installation process and `disklabel(8)` the hard drive by hand, configure the network and mount the workstation image via NFS from the build server. Finally, we run the script `getit.sh` (see Appendix A.6), which completes the installation.

6.1 Security considerations

While the installation procedure as described above is simple enough, we must to again pay attention to the security aspects of performing a network install. The same files that necessitate encryption during the update process (as explained in Section 5.2) must not be transmitted in the clear during the installation process and are therefore omitted at this point.

Since we do not intend to transmit these sensitive files, they are not stored under the `amstel:/new` directory. This in turn allows us to make the `/new` directory accessible via NFS – clearly, this would be impossible if these files resided therein: the adversary could presumably connect a rogue machine to the network and mount the directory via NFS by spoofing one of the IP addresses which are granted access. Without any sensitive information under this hierarchy, the adversary can only gain access to the same files she could otherwise retrieve from any public workstation.

On the other hand, the fact that we must not transfer the pre-shared secrets and other important files in the clear leaves us with a bit of a conundrum: the installation cannot be completely unattended, as both the sshd keys and the pre-shared secret for IKE must not go across the network unencrypted and the installation process cannot be encrypted, as the pre-shared secrets are not yet on the to-be-installed client. We will look at possible solutions to this problem in Section 7.2. For the time being, this forces us to add manually one of the pre-shared secrets after the installation process and run a partial push (to synchronize the other sensitive files) before deployment of the new host.

type	remote shell	time
minor update	rsh + ipsec	36m
minor update	ssh	49m
major update	rsh + ipsec	47m
major update	ssh	51m
full update	rsh + ipsec	149m
full update	ssh	155m

Figure 3: Scalability of a full push

6.2 Scalability

As we have seen above, installing a single host does not involve much effort, but how long does it actually take, and how well does this system scale if a large number of hosts need to be installed? The process of generating a new configuration for a new host is obviously simple and will take negligible time, as only a single short shell script needs to be executed. For a large number of new hosts, this can be wrapped into a for-loop and still not take much longer than it takes to actually enter the data the script requires. Should it be necessary to integrate a significant number of new hosts into this setup, it might be wise to collect the required information beforehand and tweak the script to run non-interactive.

The most time is spent actually installing the software on the new machine. Due to the large size of the workstation image based on the huge number of packages required, it quickly becomes clear that at this time the bottleneck lies in the network installation procedure. Most installations are currently done over the regular 10/100 campus network, as gigabit networking is not currently available in all parts of the campus. On average, the installation process from start to finish (including the creation of a new configuration on the build host, booting the new host from installation media, `disklabel(8)`ing the hard drive, running the install script over NFS) takes approximately 67 minutes. In Section 7.3 we will consider ways to improve this process.

Installing multiple hosts at the same time faces the same limitations as running multiple pushes at the same time: the more clients install in parallel, the higher the load on the build server and the higher the saturation of the network connection. Fortunately, at the moment there rarely is a need to perform more than just a handful of installations at once.

7 What's next?

While the current configuration allows for convenient and easy installation and maintenance of the workstations, like any other piece of software or administrative setup, there is, of course, always room for improvement. In this section, we will try to look at solutions for the problems mentioned in previous sections, as well as consider new features that might be desirable.

7.1 Improving the general setup

Given the large number of applications under pkgsrc control, it is crucial that the package management system works flawlessly when updating packages. Until the beginning of the year, the ever-changing nature of pkgsrc was somewhat of a problem: trying to follow a fast-moving target, our pkgsrc trees would have to be updated frequently to keep up with the infrastructure changes under `pkgsrc/mk`, but on the other hand it was risky to update the entire tree: it would then always contain the latest release of each version of software, so that building one package may pull in an update of another, already-installed package even though that is not strictly required. Rebuilding large parts of the installed software could occasionally lead to a broken dependency, leaving the workstation image in a non-stable state or – in the worst case – without a particular required package.

Fortunately, the NetBSD Packages team introduced the concept of stable pkgsrc branches, which made a lot of system administrators very happy by allowing for much easier tracking of the installed packages and keeping up with security issues. While this has improved the situation immensely, from time to time we still encounter instances of package dependencies[6] that are not strictly necessary, which is why we must maintain our own pkgsrc tree alongside the stable branch, merging changes and updates by hand. In addition, it is occasionally necessary to install conflicting packages on the same system.[7] This currently requires more modifications to the packages in question which must be remembered when updating the pkgsrc trees.

The *pkgviews* framework, introduced to pkgsrc early this year, promises to solve a number of these problems (refer to [6] for details), but unfortunately it will require starting with a clean slate, so to speak: the need to uninstall every single package and rebuild them using pkgviews is a project that we originally intended to approach during the summer of 2004, but for which, in the end did not have time. The conversion to pkgviews therefore remains on our list of improvements of the system.

As mentioned previously, all software is compiled in a sandbox and binary packages built, which are then used to maintain the actual workstation image. This is done only on an as-needed basis at the moment, allowing for the potential risk of encountering a failure at a crucial time. To avoid this scenario, it might be desirable to create a system that periodically builds binary packages from scratch, so that newer versions of binary packages compiled for our systems are *immediately* available. It might be feasible to perform complete bulk-builds to anticipate new software requests, or we might consider only building what is currently installed. Frameworks for both approaches do already exist within pkgsrc and we should be able to implement such a solution with relative ease.

While we use a problem report system at Stevens Institute of Technology to track and analyze software requests and complications, this system often relies on the user to provide an accurate analysis of the situation or detailed feedback. When changes to the software are made, they are not always annotated in great detail in the PR system, so that several weeks or months later it is not always clear why exactly a change was made, or what change was made to fix a particular problem. Maintaining a detailed changelog of the entire workstation image, possibly through the use of a revision control system, such as *CVS*, would prove useful.

Similarly, the scripts that maintain the system presented in this paper are currently not under any revision control either. Importing these files into a CVS repository would similarly allow us to better track the files as they are changed as well as quickly and easily determine why a certain change was made.

Finally, our system did not provide detailed documentation on how workstation installations or updates are done. Fortunately, this paper easily solves this problem, but it will require us to update it regularly to keep the documentation in sync with reality.

7.2 Improving Security

It might be worth considering identifying clients not only by hostname or IP address, but to also require their MAC address to match to introduce one more barrier for a MITM based attack. However, this approach introduces a significant administrative overhead while not gaining much more security: on the one hand, it is relatively easy for the adversary to spoof a

[6] *buildlink bumps* have proven to be particularly dangerous

[7] An example: we currently have both *print/teTeX1* and *print/teTeX* installed to allow our users to continue to use their older TeX files.

MAC address[8] just like she could spoof the IP address; on the other hand, it would require the careful logging of each machine's ethernet interface change. Since we routinely swap hardware among different clients, such a MAC-address-to-hostname mapping could easily become outdated.

Of more urgency would be the development of an encrypted installation mechanism, which would allow us to move to an entirely unattended install process. Considering the chicken-and-the-egg problem eluded to in Section 6.1, in this context one possible solution might be to use asymmetric cryptography rather than the current symmetric approach, which necessitates the existence of a pre-shared secret on both sides. An alternative solution might involve a custom install CD containing a special "install-key" and to perform encryption and decryption for the few sensitive files using PGP or OpenSSL. Finally, we might decide to pay the price of convenience[9] and perform installations through a private network only.

7.3 Improving Scalability

As mentioned above, the Stevens campus network does not currently provide gigabit networking. However, our build server is equipped with a gigabit network card and connected to a small private gigabit switch through which it performs regular backups of various other servers. In order to allow for a faster installation, we could perform this process over the private gigabit network, taking advantage of a more secure and much faster connection to the server.

Another way to cut down on the time taken for a full installation would be to install only what is necessary to deploy the system and then update the rest on the next push. For example, the `/usr/src` hierarchy does not need to be immediately available and could be excluded during installation. On the other hand, this increases the load during the first push, so we really do not gain very much.

The numbers in Figure 3 represent a push of all machines while synchronizing approximately 20 hosts at the same time. This number was chosen without much empirical evidence of a performance increase, but was rather based on the increase of processing speed and memory in the server compared after an upgrade. It might be desirable to perform more accurate benchmark tests and determine the appropriate number of parallel pushes to optimize CPU-, memory- and network utilization.

[8] It is worth noting that since many of our workstations are in public laboratories, it would even be possible for the adversary to simply steal the actual ethernet device of one of the machines.

[9] I.e. we no longer would be able to install a new machine from any department.

8 Conclusion

In this paper I hope to have debunked the common misconception that NetBSD is "not ready for the desktop" and stressed the importance of an "admin-friendly" operating system for production use in an environment that at the same time demands a sophisticated environment for computer specialists as well as a more traditional, user-friendly setup for novices. The desktop workstation control system presented allows, as we have seen, for simple yet scalable maintenance of a large number of identical workstations from a central build server.

The update and installation processes provide enough flexibility to allow for multiple configuration differences among hosts and subnets while at the same time ensuring that security relevant files are not compromised. As mentioned above, the infrastructure has been in production use for several years and has in addition been used as a basis of a very similar system for the maintenance of one of our High Performance Computing Facilities (also entirely based on NetBSD).

Like all software systems, there is room for improvement, and I have elaborated on a few shortcomings and considered ways to improve the system. The most important scripts of this setup can be found in the Appendix and have been placed in the public domain – any corrections, suggestions or questions are most welcome and can be directed at the author.

9 Author Information

Jan Schaumann is a System Administrator in the Department of Computer Science at Stevens Institute of Technology in Hoboken, NJ, USA, where he manages a large, nearly homogenous NetBSD environment, ports and maintains NetBSD pkgsrc tools and packages on non-NetBSD platforms such as IRIX and Linux, and teaches classes in UNIX programming and System Administration.

Jan holds Bachelor's and Master's degrees in Computer Science from Stevens Institute of Technology; he joined the NetBSD Project as a developer in January of 2002 and enjoys living with his wife Paula in New York City. The non-computer related activities he enjoys usually involve a board, often in combination with some form of H_2O. Jan can be reached at *jschauma@cs.stevens.edu*.

A Code Listings

A.1 The push.sh script used to update a single host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original authors are Thor Lancelot Simon and Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ‘‘AS IS’’ AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#set -x
#DONTDOIT=echo

LOCALFILES=/usr/local/stevens
RSYNC_RSH=rsh
PASSES=all
ROOT=/new
RSYNC_OPTS="-aSH"

usage()
{
    echo "Usage: $0: [-p <pass>] [-r <root>] [-s]"
    exit 1;
}

args=‘getopt p:r:s $*‘
if [ $? -ne 0 ]; then
    usage
fi
set -- $args

while [ $# -gt 0 ]; do
    case "$1" in
        -p)
            PASSES=$2; shift
            ;;
        -r)
            ROOT=$2; shift
            ;;
        -s)
            RSYNC_RSH=${LOCALFILES}/rsync-ssh
            ;;
        --)
```

```
            shift; break
            ;;
    esac
    shift
done

export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"

    # show when we last updated
    date > ${ROOT}/etc/updates
    echo >> ${ROOT}/etc/updates
    cat ${LOCALFILES}/UPDATES >> /new/etc/updates

    # set up exclusions.  Rather ghastly.
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/\")) print \$4}" < special.files > /tmp/x0.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/etc\")) print \$4}" < special.files > /tmp/x1.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\")  &&  \
    (\$3 == \"/usr/pkg\")) print \$4}" < special.files > /tmp/x2.$i.$$
    awk "/^EXCLUDE/ {if ((\$2 == \"$i\") &&  \
    (\$3 == \"/var/db/pkg\")) print \$4}" < special.files > /tmp/x3.$i.$$

    # start the bombardment
    if [ -e ./beforesync ]; then
        $DONTDOIT rsync ./beforesync $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/beforesync
        $DONTDOIT $RSYNC_RSH $i /tmp/beforesync
    fi

    # note that these "passes" correspond to the exclusion statements above
    if [ "$PASSES" = "/" -o "$PASSES" = "all" ]; then
        echo "pass 0: /"
         $DONTDOIT rsync ${RSYNC_OPTS} --delete --one-file-system       \
            --exclude-from=${LOCALFILES}/exclude-from           \
            --exclude-from=/tmp/x0.$i.$$ ${ROOT}/ $i:/
    fi

    if [ "$PASSES" = "/etc" -o "$PASSES" = "all" ]; then
        echo "pass 1: /etc"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete                          \
            --exclude-from=${LOCALFILES}/exclude-from-etc     \
            --exclude-from=/tmp/x1.$i.$$ ${ROOT}/etc/ $i:/etc
        $DONTDOIT rsync ${RSYNC_OPTS} ${LOCALFILES}/client-etcs/common-files/ $i:/etc
        $DONTDOIT rsync ${RSYNC_OPTS} ${LOCALFILES}/client-etcs/$i-etc/ $i:/etc
    fi

    if [ "$PASSES" = "/usr/pkg" -o "$PASSES" = "all" ]; then
        echo "pass 2: /usr/pkg"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete --exclude-from=/tmp/x2.$i.$$     \
```

```
            ${ROOT}/usr/pkg/ $i:/usr/pkg

        echo "pass 3: /var/db/pkg"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete --exclude-from=/tmp/x3.$i.$$     \
            ${ROOT}/var/db/pkg/ $i:/var/db/pkg
    fi


    if [ "$PASSES" = "/usr/local" ]; then
        echo "pass: /usr/local"
        $DONTDOIT rsync ${RSYNC_OPTS} --delete ${ROOT}/usr/local/ $i:/usr/local
    fi

    if [ "$PASSES" = "/usr/src" -o "$PASSES" = "all" ]; then

      if [ ! `grep $i small` ]; then
          echo "pass 3.5: /usr/src"
          $DONTDOIT rsync --exclude-from=${LOCALFILES}/exclude-from-src \
              --delete-excluded ${RSYNC_OPTS} --delete ${ROOT}/usr/src/ $i:/usr/src
      fi
    fi

    if [ "$PASSES" = "absolute" -o "$PASSES" = "all" ]; then
        # "pass 4": special "absolute" copies (shudder in fear!)
        echo "pass 4: absolute copies (if any)"
        $DONTDOIT eval `awk "/^ABSCOPY/ {if (\\$2 == \\"$i\\")         \
            print \\"rsync ${RSYNC_OPTS} \\" \\$4 \\" \\" \\$2 \\":\\"  \
            \\$3 \\";\\"}" < special.files`
    fi

    if [ -f specials/aftersync.$i ]; then
        $DONTDOIT rsync specials/aftersync.$i $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/aftersync.$i
        $DONTDOIT $RSYNC_RSH $i /tmp/aftersync.$i
    fi

    if [ -e ./aftersync ]; then
        $DONTDOIT rsync ./aftersync $i:/tmp
        $DONTDOIT $RSYNC_RSH $i chmod u+x /tmp/aftersync
        $DONTDOIT $RSYNC_RSH $i /tmp/aftersync
    fi

    rm /tmp/x0.$i.$$
    rm /tmp/x1.$i.$$
    rm /tmp/x2.$i.$$
    rm /tmp/x3.$i.$$

done
```

A.2 The push.batch.sh script used to update all hosts in parallel

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

BASE=/usr/local/stevens

if [ -f ${BASE}/dont ]; then
    echo "You probably don’t want to do this right now..."
    cat ${BASE}/dont
    exit 1
fi

MACHINES=‘egrep -v ^# ${BASE}/machines.${1:-all}‘

if [ $# -gt 0 ]; then
    unset MACHINES
    for SET in $@; do
        ADDMACH=‘egrep -v ^# ${BASE}/machines.${SET}‘
        MACHINES="$ADDMACH $MACHINES"
    done
fi

echo $MACHINES | xargs -n 3 ${BASE}/bgsync.sh
```

A.3 The pushfile.sh script used to update a single file

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#DONTDOIT=echo
file=${FILE:?"No file?"}

if [ -z ${RSYNC_RSH} ]; then
    RSYNC_RSH=rsh
fi
#export RSYNC_RSH=/usr/local/stevens/rsync-ssh
export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"

    $DONTDOIT rsync /new/$file $i:/$file
done
```

A.4 The runcmd.sh script used to run a command on the remote host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED
# WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

DONTDOIT=echo
cmd=${CMD:-"uname -a"}

if [ -z ${RSYNC_RSH} ]; then
     RSYNC_RSH=rsh
fi

export RSYNC_RSH

for i in $*; do

    echo "NEW HOST: $i"
    $DONTDOIT $RSYNC_RSH $i $cmd

done
```

A.5 The gen-conf.sh script used to add a new host

```
#! /bin/sh
#
# This file is placed into the public domain.
#
# The original authors are Thor Lancelot Simon and Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ‘‘AS IS’’ AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


ENVBASE=/new

if [ $1 ]; then
    NEWHOST=$1
    echo using host $1
    shift
else
    read -p "Name of new host? " NEWHOST
fi

LOCALFILES=/usr/local/stevens/
NEWETC=${LOCALFILES}/client-etcs/${NEWHOST}-etc

if [ -e ${NEWETC} ]; then
  echo "I’m sorry, ${NEWETC} already exists.  You can’t name a host ${NEWHOST}." >/dev/std
  exit 1
fi

if [ $1 ]; then
    FQDN=$1
    echo using FQDN $1
    shift
else
    read -p "Fully-qualified domain name of new host? " FQDN
fi

read -p "Enter name of primary ethernet interface (e.g. fxp0): " ETHERTYPE
read -p "Enter IP address of primary ethernet interface: " INETADDR

SUBNET=‘echo ${INETADDR} | awk ’BEGIN {FS="."; OFS="."} {print $3}’‘
DEFROUTER=‘echo ${INETADDR} | awk ’BEGIN {FS="."; OFS="."} {print $1,$2,$3,"1"}’‘

mkdir -p ${NEWETC}/ssh ${NEWETC}/racoon ${NEWETC}/ssh
cp ${LOCALFILES}/rc.conf.default ${NEWETC}
```

```
echo hostname\=${FQDN} >> ${NEWETC}/rc.conf
echo ifconfig_${ETHERTYPE}\=\"inet ${INETADDR} netmask 255.255.255.0\" >> ${NEWETC}/rc.con
echo defaultroute\=${DEFROUTER} >> ${NEWETC}/rc.conf

ln -s printcap.${SUBNET} ${NEWETC}/printcap

echo "Now you must select an X-Windows configuration for the new machine."

while [ ! -f ${NEWETC}/X11/${WHICHX} ]; do
  echo "These are the configurations available.  Please select one."
  (cd ${NEWETC}/X11; ls XF86*)
  read -p "Which configuration file? " WHICHX
done

ln -s ${WHICHX} ${NEWETC}/X11/XF86Config


echo Now you must set a root password for the new machine.

# get default, just to make sure
cp ${LOCALFILES}/master.passwd.default ${ENVBASE}/etc/master.passwd

chroot ${ENVBASE} /usr/bin/passwd -l root

# put in place
mv ${ENVBASE}/etc/master.passwd ${ENVBASE}/etc/spwd.db ${NEWETC}/

# overwrite, just to make sure
cp ${LOCALFILES}/master.passwd.default ${ENVBASE}/etc/master.passwd


RACKEY=‘hexdump -n 16 -e \"%08x%08x%08x%08x\\\n\" /dev/urandom‘;
echo -e "${INETADDR}\t${RACKEY}" >> /etc/racoon/psk.txt
echo -e "155.246.89.68\t${RACKEY}" > ${NEWETC}/racoon/psk.txt
chmod 0600 ${NEWETC}/racoon/psk.txt


echo Making SSH host keys for ${NEWHOST}...

umask 022
/usr/bin/ssh-keygen -t rsa1 -b 1024 -f ${NEWETC}/ssh/ssh_host_key -N ’’
/usr/bin/ssh-keygen -t dsa -f ${NEWETC}/ssh/ssh_host_dsa_key -N ’’
/usr/bin/ssh-keygen -t rsa -f ${NEWETC}/ssh/ssh_host_rsa_key -N ’’
/usr/bin/ssh-keygen -t rsa1 -b 1024 -f ${NEWETC}/ssh/ssh_host_key.admin -N ’’
/usr/bin/ssh-keygen -t dsa -f ${NEWETC}/ssh/ssh_host_dsa_key.admin -N ’’
/usr/bin/ssh-keygen -t rsa -f ${NEWETC}/ssh/ssh_host_rsa_key.admin -N ’’


echo I have set up ${NEWETC} for you, but you will probably need to
echo check the following files: ${NEWETC}/rc.conf,
echo ${NEWETC}/inetd.conf, $NETWETC/ipsec.conf, ${NEWETC}/racoon/psk.txt
echo
echo Remember to restart racoon and ipsec as well.
```

A.6 The getit.sh script used to install a new host

```
#!/bin/sh
#
# This file is placed into the public domain.
#
# The original author is Jan Schaumann.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHORS ‘‘AS IS’’ AND ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
# EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#DONT="echo"

for i in ‘cat /mnt2/getit.excludes‘; do
        export EXCLUDES="-s ,^./$i,,p ${EXCLUDES}"
done

PAX="pax ${EXCLUDES} -rwvp e "
DEVICE="wd0a"

echo Stage 1: Creating filesystem...
echo --------------------------------
$DONT /sbin/newfs /dev/r${DEVICE}

echo Stage 2: Mounting filesystem...
echo --------------------------------
$DONT /sbin/mount -o async /dev/${DEVICE} /mnt

$DONT mkdir -p /mnt/tmp
$DONT chmod 1777 /mnt/tmp
$DONT export TMPDIR=/mnt/tmp

echo Stage 3: Starting the bombardment...
echo -----------------------------------

cd /mnt2 && $DONT $PAX . /mnt/

cd /mnt2/usr/mdec
$DONT ./installboot -v biosboot.sym /dev/$DEVICE

echo "Remember to:"
echo "    - manually install etc/racoon/psk.txt"
echo "    - reboot, ifconfig and push"
```

References

[1] *Documentation on the NetBSD Package System* Hubert Feyrer, Alistair Crooks et al, pkgsrc/Packages.txt, October 2004

[2] *The NetBSD Packages Collection*
`http://www.NetBSD.org/Documentation/software/packages.html`

[3] *Binary emulation in NetBSD*
`http://www.NetBSD.org/Documentation/compat.html`

[4] *High Performance Computing at Stevens Institute of Technology*
`http://www.cs.stevens.edu/~jschauma/hpcf/`

[5] *The NetBSD Operating System* Federico Lupi et al, October 2004
`http://www.NetBSD.org/guide/en/`

[6] *Package Views - a more flexible infrastructure for third-party software,* Alistair Crooks, November 2002
`http://www.NetBSD.org/Documentation/software/pkgviews.pdf`