Installing and maintaining clusters of UNIX servers using PXE and rsync

1. Introduction

If you've ever worked at an ISP, much of the start of this paper will probably look very familiar. It is written from the perspective of XS4ALL since that's the ISP I'm most familiar with, but a lot of the issues will hit home to many sysadmins that have managed computer systems at a medium sized ISP. We've all faced similar problems. How to provide reliable service for your customers, usually on a low budget. This paper is about how XS4ALL faced some of these problems and found a solution that worked for us.

Started in 1993, XS4ALL was one of the first Internet providers in Europe. We started in a period when the hype had not really begun yet. The times before the web. Before dotcoms. Our customers were mostly technically advanced, and didn't expect very much service. If something went down, they would just try again later. There was just one server, on a simple 10mbit network, connected to one router. The router had a 24.4k modem link. Life was simple.

This wouldn't last for long. As the Internet gained in popularity, so increased the number of servers. And with the number of servers, increased the complexity of the network. [1] As the number of servers grows, you start hitting limits of the current setup, and have to think of new ways to handle things. First we had one server. Then we needed a second one and split up the services, then another, a second Internet link, separate mail servers, you name it. And then we didn't just need one of each, but two or more. But still it remained manageable; the number of servers could be counted on two hands. Most Internet providers, especially the earlier one, followed similar paths. From humble beginnings, growing to meet the needs of the customers.

The Dark Ages

As XS4ALL started adding servers, not much thought was given to how these servers were installed. There usually wasn't time, because we needed the new server yesterday, not tomorrow. We started with PC hardware, then had a short period of using Sun, and finally returned back to PCs, using mostly BSD/OS.

We still didn't think much about how we installed servers. If we needed a new one, we'd get the latest CD and do an install. Since all installs were manual, they all ended

[1] As an interesting note, the first thing we had to split away from our one server was Usenet. Even then Usenet created a problem. Some things never change. Even now, Usenet is one of the most problematic things an ISP faces. You can fill a whole paper on that subject alone.

up different. It worked, but it created quite a few problems over the years. As we reached the several dozen server stage, we had a multitude of different OS versions. And even if the version was the same, the configuration was usually different. We also started using Linux for some services, which only increased the problem. We had created a monster.

Recognizing the problems

The problems we were having can be put into several categories.

- Installation.
- Maintenance
- Security
- Upgrades

Installation.

Installations were very time consuming. You'd have to sit in front of the server, answering installation questions as the CD does its install. Everyone had their own way of doing things, and even if you tried, you couldn't get everyone to follow a specific setup. Once the setup was finished, you'd have to post-configure the box. This was also a manual process, in which you tried to remember the steps you took last time. Things were often forgotten, resulting in operational problems

Maintenance

As one can imagine, maintaining this monster was a nightmare. Every server was different, and you didn't know what was different until you went and looked. New versions of applications had to be compiled for every box separately. More often than not a few servers were forgotten.

What also didn't help maintenance was our unconventional naming scheme. We started using names from the comic Asterix and Obelix. Funny at first, but when you have 30 servers all ending in "ix", it becomes quite annoying. It made the learning curve of new sysadmins steeper then necessary because they had to find their way along the Fistix, Doetnix, Geriatrix, Majestix and Getafix.

Security

For XS4ALL, this especially was an important issue. Trying to keep all these boxes secure was increasingly difficult as the attacks became more elaborate. Luckily our roots are in the hacker scene, so we would often get advance warning of holes. And if someone got in, our open way of dealing with it usually meant no serious harm was done. But it was a ticking time bomb, waiting to explode.

A specific side issue to consider is what I call "orphaning" a server. You install a server, configure it and turn it on. And it works so well, you never look at it anymore and basically forget about it. Most of the security compromises at XS4ALL happened because the server in question was forgotten about.[2]

Upgrades

There are three kinds of upgrades to consider. First there are OS upgrades. We tried to avoid these as much as possible. Simply because they were so much work and we couldn't spare the time. Every box had to be manually upgraded and then extensively tested.

Then there's the application upgrades. We of course had to do these as new options became available, or security problems got fixed. But again, every upgrade posed problems and was time consuming.

Last there's the hardware upgrades. Usually they were easy, but as some of our servers had pretty old software installations due to failure to upgrade, the hardware wasn't always supported. This often meant we scavenged hardware off of retired servers to keep other servers running.

Besides these main problems, there were other things to consider. New employees often took a long time to understand the whole setup and the relationship between servers. "What does Obelix do?" And not just new employees were confused about things. Even people working here for years often didn't know what some part of the system was doing.

The root of all of these problems is of course in documentation. But since every server was different, documentation was a real problem. Every box needed its own documentation, which was so much hassle we basically didn't do any documentation[3]. It was a negative spiral making things worse and worse.

A new hope

Seeing all these problems you'll probably wonder why we didn't do anything about it all these years. Actually, things were not that bad. We managed to keep most of

[2] An interesting example of this case is one my colleague Jan- Pieter Cornet did a talk about at HAL2001. He talked about how we discovered and tracked a hacker. He didn't mention they hacked our secondary DNS server, which is fully automated, and no one ever bothered to check it and fix known bugs. We forgot about it. http://www.hal2001.org/hal/03Topics/speakerlist.html#cornet

[3] The one exception was the network documentation. As we entered a new system room, the need for correct documentation was apparent to everyone. Ever since day one we managed to keep correct documentation for every Ethernet cable, console port, fiber cable, rack position etc. It was one of the first steps to realizing things had to change. People that forget to document are quickly ridiculed and put in their place by everyone. Nothing works as good as peer pressure :)

these problems invisible to customers. We hadn't reached the limit yet of what we could manage this way and were doing quite well even with all these problems.

Two things coincided during 2000 that made us rethink what we were doing.
First we were about to offer a new service, Dedicated Hosting, which would have us install and maintain potentially dozens, maybe even hundreds of new servers. Customers got their own box, with or without server management.

Second, there was SANE2000. At SANE we saw a talk by Joe Greco about setting up a news service using Diablo. Since we wanted to upgrade our news servers, we went to SANE to see his talk. In the weeks after SANE we implemented a news service based on this talk, which included the use of FreeBSD. An OS we hadn't really used before but was intricately linked to Diablo at that time.

We liked what we saw with FreeBSD. It wasn't that much different from BSD/OS, but free, and it had a nice ports system making installing software a breeze. After a few months of running FreeBSD on our news systems, the most powerful cluster of servers at XS4ALL, and doing a great job at it, we chose to use FreeBSD for all our servers.

In the following months, the Dedicated Hosting service was also taking off. We had already chosen FreeBSD for it, but were still doing it the manual way. Installing from CD and adding packages to the base install. We realized this had to change. It was just too expensive to manage these boxes separately. We needed to automate the installs and then centralize the maintenance.

One of the first methods recommended to us for installing FreeBSD servers was PXE. It was used with success at several ISPs and large content providers. It quickly became apparent that a PXE based install was exactly what we were looking for. It only took a few days to create a working PXE based installation process, producing identical servers. It was much faster than any CD installation, so we decided to stop looking at anything else.

The next step

Creating identical servers was only a first step towards fixing and preventing our problems. The biggest challenge was still ahead, making sure these servers remained identical. We had already used rsync to keep parts of some old BSD/OS servers in sync, so we decided to just use rsync to keep these FreeBSD in sync 'while looking for something better'.

But 'something better' was not necessary. Using a master server we kept the entire Dedicated Hosting cluster in sync without any problems. Over time this grew into a site-wide deployment of rsync based maintenance. The next chapters will explain the details of our work. First I'll explain our PXE based installations, and then our rsync based maintenance.

2. PXE

PXE stands for Preboot Execution Environment. It is a standard created by Intel specifying the functionality needed for a boot ROM to boot over the network. It is built on a foundation of already existing Internet Protocols, namely TCP/IP, DHCP and TFTP.

The DHCP protocol is extended using vendor options, which is allowed by the DHCP standard, and does not disrupt normal DHCP operations. You will need a DHCP server that understands these extensions. You also need a network interface (NIC) that has been outfitted with PXE client software.

This allows a computer (the client) to get an IP number from a DHCP server, download a bootstrap program using TFTP and perform specific tasks based on operator configuration.

Some of these tasks include:

- Remote boot. This is the classic situation of allowing a diskless server to boot from the network.
- Remote Emergency boot. If a computer fails to boot due to hardware or software failures it can use a network boot image to perform notification or diagnosis.
- Remote installation. If there is no OS installed on a computer outfitted with a harddisk PXE can be used to automate OS installation and configuration.

PXE protocol details

Explaining the full details of PXE is beyond the scope of this paper. I will briefly discuss some of the steps in the PXE protocol, to better understand how this can be used to install servers. I will also assume the use of FreeBSD.

The BIOS of a PC contains several boot options. You can choose to boot from a HD, a floppy, a CD-ROM, or in more recent BIOSes from a Network Boot device. These are usually based on PXE capable NICs.If you want to boot from the network, you will of course need to be connected to a network.

When the BIOS selects the Network Boot option, the NIC broadcasts a DHCPDISCOVER packet on the network, including PXE specific vendor extensions. If a DHCP server that understands these PXE extensions is available, the DHCP server responds with a DHCPOFFER packet containing an IP number. (If there is no PXE capable DHCP server available the process will timeout and eventually the BIOS will select another boot device)

The client then sends a DHCPREQUEST packet (as a broadcast or directly to the DHCP server). The DHCP server responds with a DHCPACK packet that contains a TFTP Boot Server Name, a Boot File Name and a Root File system location.[4]

The client now connects to the TFTP server and downloads the boot image.

Finally the client executes the boot image. This image is a normal bootloader, with the exception that it understands two directives provided by the DHCP server. These are “next server” and “option root-path” which tell the bootloader the server and path to a remote directory it will try to NFS mount. This directory should contain a kernel, a /boot directory with second and third stage bootloaders and several config files.

Installing FreeBSD using PXE

Our goal was to use PXE to create an automated FreeBSD install, resulting in identical servers. To accomplish this, several conditions have to be met.

We need:

- PC with a PXE capable network card and suitable harddisk.
- Ethernet connection.
- DHCP server understanding PXE extensions.
- TFTP server.
    - ✓ Boot image
- NFS server.
    - ✓ Boot loader files
    - ✓ Loader configuration
    - ✓ Kernel.
    - ✓ Memory file system
        - ▪ Install.cfg
- Copy of FreeBSD Install files.

- <u>PC with a PXE capable network card and suitable harddisk.</u>

This can basically be any type of PC, as long as it has a network card with PXE boot ROMs. It is very advisable though to, just like your OS installations, also standardize your hardware. Therefore I’m going to elaborate a bit on this point.

[4] This is a highly simplified description of the actual process using FreeBSD. In reality more steps are required. See the PXE specs for more detail.

Not standardizing your hardware has some of the same problems as those seen with not standardizing your software. You'll run into all kinds of installation problems, maintenance problems, upgrade problems etc. Spare policies alone are a nightmare to consider.

Assuming PC hardware, it is recommended to use 1U or 2U rack mounted PC cases. This has become a custom, especially in Data Centers where rack space is usually the cost driving factor. Use more space and you pay more. Combine this with motherboards that integrate 10/100/1000 mbit Ethernet, VGA and SCSI or IDE.

I would also recommend the use of SCA capable cases. It makes swapping harddisks a breeze. Especially if you make sure your harddisks are also pretty much standardized. We use harddisks of one specific brand only, moving to newer versions as they become available.

Some of the advantages:

- 1U/2U cases save lots of space. We have over 30 PCs in some of our 19" cabinets. Just make sure you have enough cooling. Fortunately these cases don't generate a lot of heat.
- Spare policies are simple. Don't bother with replacing or repairing individual cards. Replace the whole box. Just keep as many empty boxes available as you think are necessary. We keep 1 or 2 of each type of server.
- Support for most Operating Systems is excellent.
- It looks cool!

The motherboards we currently use have two onboard PXE capable network cards.

- Ethernet connection

Even though it's obvious, it had better be said. You will need an Ethernet connection to connect to the different types of servers.

Since our workbenches are used for several types of installs, including several manual installs, we use specific UTP cable colors for our 'PXE installations'. This might seem superfluous, but if a BIOS has PXE configured as it's primary boot option, it will boot using PXE if it can, and in this case automatically install FreeBSD.

For this reason I recommend you always split your PXE installation network from other networks, because if some desktop user fiddles with their BIOS settings choosing PXE as primary boot, they'll be very surprised to find themselves with a FreeBSD desktop. This "Windows Upgrade" is probably not appreciated by everyone.

- **<u>DHCP server</u>**

You will need a DHCP server that understands PXE extensions. You can use the ISC DHCP server, which comes with most free OS versions.

All you need to do to configure an ISC DHCP server is add a few lines to most default configurations. A DHCP config could look like this:

```
server-name "DHCPserver";
default-lease-time 86400;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option domain-name "xs4all.nl";
option domain-nameservers 194.109.6.66,194.109.9.99;
option routers 192.168.1.2;
subnet 192.168.1.0 netmask 255.255.255.0 {
        range dynamic-bootp 192.168.1.10 192.168.1.254;
        filename "pxeboot";
        next-server 192.168.1.3;
        option root-path "/usr/local/export/pxe";
}
```

This configures the DHCP server to hand out IP numbers between 192.168.1.10 and 192.168.1.254. There are only 3 additional lines necessary to configure this for PXE use.

```
    filename "pxeboot";
```

This tells the PXE client that the boot image filename is “pxeboot”.

```
    next-server 192.168.1.3;
```

This is the IP number of the NFS server used to find the additional necessary files

```
    option root-path “/usr/local/export/pxe”
```

This is the directory on the NFS server where the boot image can find additional bootloaders, kernel and configuration files.

- **<u>TFTP server</u>**

Almost every OS already has this available. Usually all that’s required is to turn it on in the inetd config file.

pxeboot

The PXE boot process needs to grab a boot image from this TFTP server, which we named “pxeboot” in the DHCP config file. So we’ll need to put a boot image file in /tftpboot. Luckily FreeBSD comes with the necessary PXE capable boot image file. It is named “pxeboot” and can usually be found in the /boot directory.

Copy this file to /tftpboot and the TFTP server is all set.

- <u>**Copy of FreeBSD install files**</u>

Since our goal is to install FreeBSD on this machine, we’ll need a copy of the FreeBSD distribution that can be NFS mounted. Since we’ll be setting up an NFS server in the next section, easiest would be to copy the contents of the FreeBSD install CD to a directory on the NFS server and export that directory along with the “option root-path” directory. You could even make it a subdirectory of the root-path directory so you only need to export one directory.

- <u>**NFS server**</u>

We need the NFS server so “pxeboot”, the boot image, can locate additional bootloaders, kernel and config files. In addition to this we’ll also put a copy of FreeBSD on it as mentioned before.

Setting up the NFS server is pretty straightforward. You need to export the directory used in “option root-path”. Then restart or start the NFS server. Under FreeBSD this means starting nfsd and mountd.

The bootloader files

To continue the boot process, “pxeboot” needs additional boot files. The boot files need to be in a directory “boot”, inside the directory configured as “root-path”.

Create that directory (mkdir /usr/local/export/pxe/boot) and copy the files “loader”, “boot1” and “boot2” from /boot to that directory.

Loader configuration files

The program “loader” is the final stage in the boot process. It can be scripted to automate certain tasks. We need to do this, so we’ll need to create a config file for it. This is done using the file “loader.rc”.

A simple example looks like this:

```
echo Loading Kernel...
load /kernel
set choice=default
echo
echo Please select one of the following installs:
echo
echo default
echo scsi
echo dh
echo
read -t 15 -p "Enter your choice: " choice
echo
include /boot/loader.rc.$choice
echo booting...
set vfs.root.mountfrom="ufs:/dev/md0c"
boot
```

And loader.rc.default as an example*:*

```
load -t mfs_root /mfsroot-default
```

This loads a kernel, sets a default installation choice, gives the sysadmin the option to choose a specific installation, loads the chosen memory file system using the include file (more on this later), tells the kernel to use the memory file system as its root file system, and then boots with that kernel.

Kernel

As said before, the loader needs a kernel to boot into. You can use the GENERIC kernel, or create one yourself. If you create one yourself, make sure MFS (Memory file system) support is compiled in.

Copy the kernel to /usr/local/export/pxe. Make sure it's called "kernel".

Memory file system

When the kernel has finished loading, it needs a file system to continue. Generally this would be a file system on a harddisk. But because we don't have a usable harddisk yet, we have to provide the kernel with a file system of our own. This can be accomplished using a memory-based file system. There are several ways to create a memory file system. The easiest is to grab one from the FreeBSD distribution we already have[5]. First we need to get the mfsroot file system file from the floppy image on our local FreeBSD copy.

[5] Or check this URL: ftp://ftp.freebsd.org/pub/FreeBSD/snapshots/i386/mfsroot.flp

```
# mkdir /mnt/floppy
# vnconfig vn0      \
     /usr/local/export/freebsd/floppies/mfsroot.flp
# mount /dev/vn0 /mnt/floppy
# cp /mnt/floppy/mfsroot.gz /tmp
# umount /mnt/floppy
# vnconfig –u /dev/vn0
# gunzip /tmp/mfsroot.gz
```

Then we mount this so we can edit the contents.

```
# vnconfig vn0 /tmp/mfsroot
# mount /dev/vn0 /mnt/floppy
```

We now have a floppy sized memory file system on /mnt/floppy. Although this is probably enough space for most people, you can also create your own memory file system, and use whatever size you like. If this is enough for you, skip the following section.

Creating a Memory file system from scratch

Another way to create a Memory file system is starting from scratch. Here is what you do:

- cd to the directory you'll be exporting.
```
# cd /usr/local/export/pxe
```

- create a 10MB empty file.
```
# dd if=/dev/zero of=mfsroot bs=1k count=10000
```

- assign device vn0 to this file
```
# vnconfig -e -s labels vn0 mfsroot
```

- create a disk label for this device
```
# disklabel -r -w vn0 auto
```

- create a file system
```
# newfs /dev/vn0c
```

- and mount it
```
# mkdir -p /mnt/mfs
# mount /dev/vn0 /mnt/mfs
```

At this point you have an empty 10MB mounted memory file system on /mnt/mfs. Now we need fill it with the basic files needed for FreeBSD to continue. We can use the same floppy file system as before and copy the contents to our own file system:

- mount the floppy just like we did before

```
# mkdir /mnt/floppy
#vnconfig vn1       \
     /usr/local/export/freebsd/floppies/mfsroot.flp
# mount /dev/vn1 /mnt/floppy
# cp /mnt/floppy/mfsroot.gz /tmp
# umount /mnt/floppy
# vnconfig -u /dev/vn1
# gunzip /tmp/mfsroot.gz
# vnconfig vn1 /tmp/mfsroot
# mount /dev/vn1 /mnt/floppy
```

- using rsync we copy the contents to the 10MB file system

```
# rsync -avzH /mnt/floppy/ /mnt/mfs
```

- unmount the floppy, since we don’t need it anymore

```
# umount /mnt/floppy
# vnconfig -u vn1
```

We now have a 10MB memory file system we can use.

Install.cfg

At this point all we have is a netboot recipe booting up FreeBSD. We created the files necessary for a computer to boot up a FreeBSD kernel over the network and mount a file system. So how do we get to installing?

The file system we made lacks an /sbin/init. When FreeBSD fails to start /sbin/init, it automatically starts up sysinstall, the FreeBSD installer software. Of course that still would be useless if we then had to answer questions manually. Luckily sysinstall can be scripted using a file called “/install.cfg”.

The complete setup of the scripting language is beyond the scope of this paper. An example can be found at http://www.xs4all.nl/~scorpio/sane2002. Using this file, sysinstall will install a copy of FreeBSD on an internal harddisk.

We place this install.cfg on our mounted memory file system. We can now umount this file system and put the mfsroot file into the directory we exported on the NFS server.

As I showed in the loader.rc example before, it’s possible to choose between different memory file systems by creating a menu in the loader.rc. Each file system could contain a different install.cfg file, which can be used to install different types of servers.

Putting it all together

So let’s see what has happened so far.

The BIOS started a Network Boot. The network card, which is PXE capable, talked to our DHCP server and got the location of a boot image on a TFTP server and the location of an NFS file system.

It grabs the boot image “pxeboot” from the TFTP server, and executes it. The boot image uses the NFS file system to continue into its second and third stage. The third stage bootloader loads the kernel and tells it to use a memory file system, which is also loaded from the NFS file system.

The kernel checks the hardware, and then tries to execute /sbin/init on the memory file system. This fails, and instead sysinstall is started. Sysinstall is scripted using an install.cfg file, which tells it to install FreeBSD on a harddisk and shut down.

This whole process, from start to finish, only takes a few minutes. Moreover, using some of the features of loader.rc, we can vary different aspect of the installation process. This can be used to automatically install different types of servers.

To improve on this concept, we have adopted or are going to adopt some of these additions:

- Since our hardware is identical, we usually just plug an empty hardisk into a dedicated installation server. Turning the server on is enough to start and finish the installation process (unless you wish to change the type of server installed using a loader.rc menu). After installation the server is shutdown from install.cfg, and we can remove the harddisk and put it into a production server

- We normally keep a stack of already installed harddisks ready, to replace broken servers or to be able to quickly fire up new servers.

- One can easily install dozens of servers at the same time, by just plugging them all into power and Ethernet at the same time. The network will slow down a little, slowing the process down slightly. We often order a dozen new servers at a time, installing them all at once by just plugging them in. We then hang them into production 19” racks, waiting for new customers.

- Instead of changing the type of server by using a menu in loader.rc, you can also set up different networks, each with their own Ethernet. Mark those Ethernets with a color, and you won’t ever have to do any manual interaction. Just plugging a server into the correct Ethernet will result in a specifically created server.

3. RSYNC

The manual has this to say about rsync:

```
rsync  is a program that behaves in much the same way that
rcp does, but has many more options  and  uses  the  rsync
remote-update  protocol  to greatly speedup file transfers
when the destination file already exists.

The rsync remote-update protocol allows rsync to  transfer
just  the differences between two sets of files across the
network link, using an efficient checksum-search algorithm
described  in  the  technical report that accompanies this
package.
```

In short, it allows you to keep two sets of files synchronized over the network. This could be just a single directory, or in our case, a complete server. It also allows you to exclude some files, so you can keep some specific local configuration files separate. We had used rsync successfully for several months to keep our Dedicated Hosting cluster synchronized. This worked out better than we really expected. So good in fact, that we stopped looking for 'something better'.

As we were about to start using several other FreeBSD clusters (replacements of old BSD/OS systems) we started thinking of a way to use rsync for all our servers. Over a period of several weeks we further developed the basic idea of using a master server to keep production servers in sync, and came up with a simple solution that has since been the core of our centralized maintenance.

Clusters

At this point it is a good idea to explain the term cluster. We define it as a set of servers with the same task. As an ISP we offer different types of services. As our user base grows, we need to scale our servers to meet the demand.

One way to do this would be to buy a bigger server every time you start to hit a performance limit. This usually becomes excessively expensive. You also create huge single points of failure. We try to avoid this route.

We have chosen to scale our services by using groups of smaller servers. Every service consists of one or more servers. This keeps costs down, and budgets easy to calculate. And if you always make sure you have one or two extra servers per group available, a server failure won't hurt your performance.

Because of our choice to use groups of "smaller" servers, we have a lot of clusters. Some only have two servers; some consist of several dozen servers. Since each group has a specific task, all servers in one cluster need to be identical, and should be centrally maintained.

parent/child

Every one of our clusters has what we call a parent. The parent is basically the same as the actual production servers. There are only two real differences. First, it doesn't actually do any work. Second, because of this, it is usually low-end hardware.

The differences between the different clusters aren't that big. We use one specific OS version (in our case FreeBSD 4.2, although we're now moving to FreeBSD 4.5). And on top of that we install a specific application. You may have a Sendmail installation on one cluster to run smtp.xs4all.nl. Or you may have a specific Apache config on a hosting platform. Most of the rest of the server is really the same base OS.

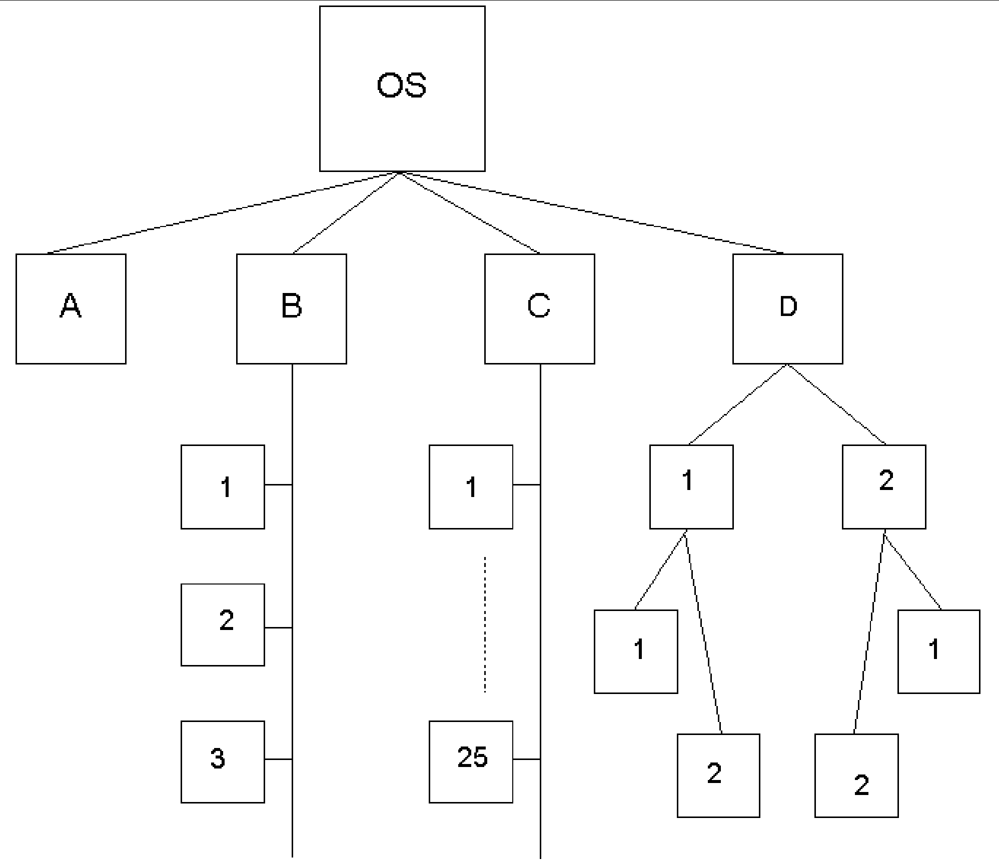
So not only are the individual servers in a cluster identical, even the clusters themselves are very much alike. We therefore created a parent for all the clusters, called the OS parent. The OS parent, in our case a server running FreeBSD 4.2, is the master of all 'standard' software installations.

This creates a tree of servers where each branch has a parent. You could have as many branches as you want, and as many levels as you want. The parent of each branch is the master server that synchronizes all the children in the level directly below it. Each of the children could itself be a parent of a new branch. At XS4ALL we normally have a maximum of 4 levels. (See figure 1).

It is of course possible to have several trees. This could happen when your have sufficiently different base installation. A good example for this would be several OS versions. We currently have two trees, one for FreeBSD 4.2 and one for FreeBSD 4.5. As we upgrade servers from 4.2 to 4.5, they move from one tree to the other. (More on upgrades later).

Naming scheme

As I mentioned in the introduction, your naming scheme can cause confusion. A good naming scheme makes is obvious what the task of a server is. A bad naming scheme would do..well, the opposite. We have chosen to name our clusters with names that define their task. For example, servers in our POP cluster are called mailpop1.xs4all.nl, mailpop2.xs4all.nl etc. Our Webmail servers are called webmail1.xs4all.nl, webmail2.xs4all.nl. You get the point.



In this example the OS master has 4 children:

- A. This child is not in itself a parent of a cluster. This could be a simple server like a tape backup server. Server A is the actual production server.
- B. Here we have a parent of three servers. Server B itself does not do any production work. The three children do the actual work. This could for instance be a cluster of SMTP servers.
- C. This cluster isn't much different from B. It just shows that it doesn't matter how many actual production servers there are. It could be hundreds. Our largest cluster currently has about 50 servers, all synchronized from 1 master server.
- D. This shows that you can have multiple levels of parents and children. Server D is the parent of 2 children. These 2 children are themselves parents of a set of production servers. These could for instance be two sets of nearly identical hosting platforms, each for a slightly different product set.

Figure 1

All servers that act as a parent to a cluster are called ‘zero’ servers. The master server to our Webmail cluster is for example called webmail0.xs4all.nl. This easily identifies all master servers. The only exception to this is the OS master. We use the OS version in its name. We currently run the servers freebsd-4.2.xs4all.nl and freebsd-4.5.xs4all.nl as OS masters.[6][7]

Testing

Sometimes changes you make might break your production servers. This is something you’d normally want to avoid. But testing on your master is also not always a good idea. As you are testing, one of your colleagues might want to update a piece of software. This could result in installing partly configured or broken software.

For this reason we created test servers. Each parent has a clone that can be used for testing. Then to actually test the changes on a production server, there is also a production test server. Once you are happy with the changes, you synchronize the parent with the clone, and then synchronize the production servers with the parent. We only use this when we aren’t confident about our changes. Normally installing a small update doesn’t need this elaborate testing. But if you for instance upgrade from Sendmail 8.10 to Sendmail 8.12, you might want to make very sure that your upgrade works.

A cluster (I wont draw the whole tree) would now look like figure 2. It is not really necessary to provide a pair of test servers for each cluster. You could even, in theory, only have two servers for testing in one tree. Before you want to test you synchronize the TEST0 server with the master server of the cluster you are working on. Since your base OS is the same everywhere in the tree, this should work fine.

Synchronizing the servers

So far I’ve mostly talked about the theory of our solution. So how does this translate to day-to-day work?

At first we manually ran rsync to synchronize two servers. This works but is prone to human error. We therefore created a perl script to make this task easier.[8] This perl script, we call it sync.pl, is run from the parent, and synchronizes one or more children.

[6] We actually started out calling our FreeBSD 4.2 master freebsd0.xs4all.nl. This was before we started upgrading to FreeBSD 4.5 ☺

[7] We are also experimenting with a Linux OS tree.

[8] See http://www.xs4all.nl/~scorpio/sane2002

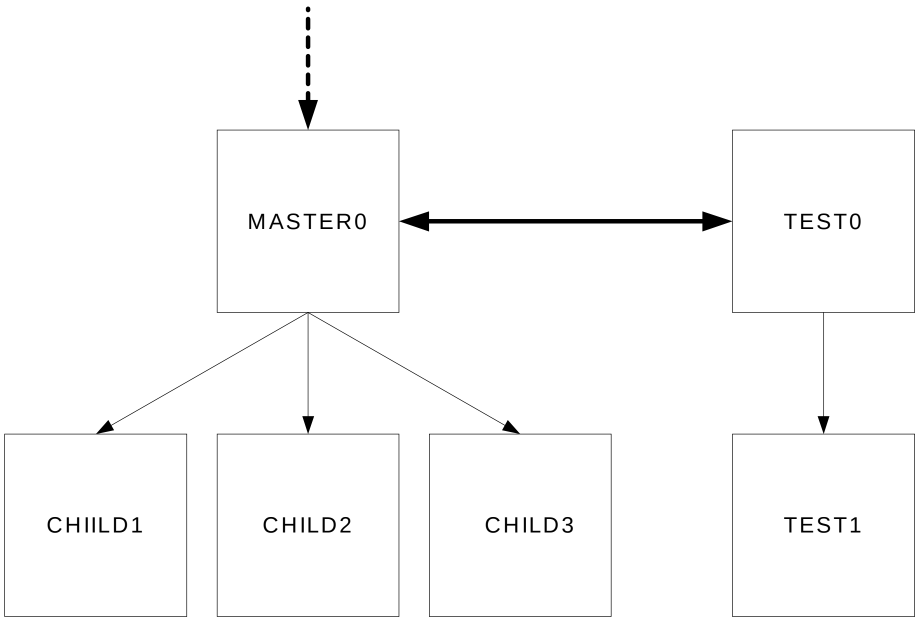


Figure 2

The syntax of sync.pl is as follows:

```
$ sync.pl

Usage: sync.pl -t -s <site> -f <conf file>

-f      alternate file. Default is sync.conf
-t      test, don't actually do it.
-s      hostname to sync. Use "all" for all sites.
-c      allow recursive updates
```

Configuration file

You normally don't want to synchronize each and every file to the children. You need to exclude certain files like configuration files containing the server's IP number (rc.conf on FreeBSD). Other files you might want to exclude are customer data directories, temporary files, log files, etc.

We maintain a configuration file containing a list of children and the exclude file to use. The sync.pl script uses the option -f to include a specific configuration file. If -f isn't provided we use the default filename "sync.conf".

This file might look like this on an OS master server (I'll explain the third parameter later):

```
$ cat sync.conf
# hostname         exclude_file               recursive?
#
smtp0.xs4all.nl     exclude.smtp              yes
webmail0.xs4all.nl  exclude.webmail           yes
dh0.xs4all.nl       exclude.dh                yes
backup.xs4all.nl    exclude.backup            no
```

The parent server needs to have an exclude file for every cluster of children. The OS master typically has an exclude file for every child, because every child is likely to be different. The parent of a specific cluster, say smtp.xs4all.nl, needs only a single exclude file because all of its children are equal. The sync.conf file for a cluster master might then look like this:

```
$ cat sync.conf
# hostname          exclude_file        recursive?
#
smtp1.xs4all.nl     exclude.smtp        no
smtp2.xs4all.nl     exclude.smtp        no
smtp3.xs4all.nl     exclude.smtp        no
smtp4.xs4all.nl     exclude.smtp        no
```

Besides the individual exclude files, we also maintain a global exclude file. This file contains exclusions that need to be done for every server.

Exclude files

For full details of the syntax of exclude files, please check the rsync manual. I will give a short explanation of how we use the exclude files.

An exclude file could look like this:

```
$ cat exclude.example
/tmp
/var/mail
/var/log
/etc/rc.conf
+ /var/db/
+ /var/db/pkg/
/var/db/*
```

This would cause the /tmp, /var/mail, /var/log, /etc/rc.conf and /var/db/ directories to be excluded from the sync. The + sign can be used to include directories or files that reside inside excluded directories. In this case we include /var/db/pkg. This example is merely an example, and not meant to be complete.

Testing a sync

The wrong rsync could wreak havoc on a server. You might have forgotten to include a customer directory in the exclude file, which then gets deleted (rsync doesn’t merely install or updates files, it also deletes files).

This can be easily avoided by using the -t option in sync.pl. This option causes rsync to go through all the motions but not actually do any changes (the -n option of rsync). It outputs a list of all the actions it’s going to perform, but not actually do them. This gives you a chance to preview all the changes and fix any problems that arise.

Updating all the children

If you have 150 children, doing them one by one is going to take a lot of time. Since they are all identical, you might as well only test one of them, and then synchronize all of them. The -s option of sync.pl is used for this. It can take either one hostname or the keyword “all”. If a single hostname is given, that host is looked up in sync.conf. The keyword “all” tells sync.pl to update all the hosts in the configuration file.

Recursion

The configuration file specifies a third parameter. This parameter enables recursion. One of the servers you are updating could itself be the parent of a cluster. This is typically the case when you install or update software on an OS master. This change needs to be performed throughout the complete tree (unless excluded in specific exclude files). You could do this by logging into each parent and performing an update, or you could enable recursion.

Recursion is enabled by using the -c option. If this option is present, and the configuration file contains “yes” in the third column, sync.pl will automatically log into this specific child and fire up a new sync.pl on this server (see the section on ssh on how this is done).

At this point recursion forces the use of ‘-s all’ on the child. It is really meant to propagate a simple change throughout the whole tree.

Examples

```
freebsd-4.2:$ sync.pl -t -s smtp0.xs4all.nl
```

Do a test sync from the OS master to smtp0.xs4all.nl. This will show all the changes that are going to be performed.

```
smtp0:$ sync.pl -t -s all
```

Do a test sync from smtp0 to all the children.

```
smtp0:$ sync.pl -s all
```

Actually do the sync to all the children

```
freebsd4-2:$ sync.pl -t -s all -c -f test.conf
```

Run a recursive test sync to all the children of the OS master (so the whole tree), using a specific configuration file named test.conf. We would normally use this after we've done a small OS change, and want to propagate this change through the whole tree. After the test looks good, we would then run the following command:

```
freebsd4-2:$ sync.pl -s all -c -f test.conf
```

SSH, passwords and security

So far I have not mentioned how rsync actually connects to the children. Normally rsync uses rsh. We prefer ssh, which can be turned on using the -e option in rsync.

You can leave it at this, and manually type in the root password every time rsync asks for it. This would certainly be most secure. It is also, unfortunately, most cumbersome. We currently have over 150 servers in one of the trees, which means an update performed on the OS master, recursively installed on all children, would have us type in more than 150 passwords. We quickly got tired of this, and opted to abandon some security for ease of use.

SSH allows you to create key pairs to authenticate logins on an SSH server. You create a public and private key using ssh-keygen. The public key is added to the file /root/.ssh/authorized_keys on the remote host. The private key is used on the parent server. The key itself is protected by a passphrase. You could leave the passphrase empty, which would make sure you don't have to type any passwords to run the sync.

Obviously this is a dangerous option. If your private key gets out, unwanted people can access your servers. And if they gain (unauthorized) access to your master server, nothing stands in the way of gaining access to all the children. Which in the case of the OS master would be all of your servers.

We have chosen a middle ground, but leaning towards ease of use. Using the key agent feature of ssh we create a temporary key repository. This way the sync.pl script only asks for a passphrase once, after which it can use this key on all the children (see the ssh manpage for more information about the use of a key agent). After finishing the sync the key agent is immediately killed.

Of course this doesn’t buy you very much security. If someone has gained access to one of your master servers, you’re already in trouble. There are some steps you can take to minimize risks:

- The authorized_keys file allows several extra parameters to control what can be done if you present the correct private key. We use 2 options to control access:
    - From=”...”. This controls which hosts are allowed to authorize against this specific public key. By only allowing the parent of this server to authorize, access to the parent is necessary to be able to login using this key.
    - Command=”...”. This controls which commands a valid key can execute after successfully being validated against this public key. By only allowing rsync to be executed, you make it slightly more difficult to hack into children once your parent server is compromised. As a more important side effect it also makes sure that only that specific command gets executed, limiting the chances you run the wrong command from the parent. The command given by this option actually replaces any command you give remotely.
- Use firewalls to control access to your servers. Especially your parent servers need to be well protected from outside access.
- Use an Intrusion Detection System to notify you of any strange behavior, especially concerning your master servers.

How recursion works

To allow for recursion, we created a specific ssh key pair. By using the “Command=” option, we start a specific program when this key is used to authenticate. This program is a simple script that fires up a new “sync.pl –s all” on this child, causing it to update all its own children.

Checking daily changes

An interesting side effect of our rsync-based system is that the system itself can be used to enhance your security. Every night we run the following command on all the master servers from cron:

```
$ sync.pl –t –s all
```

The output of this command is a list of all the changes on all the children of this specific master server. If there are any changes to be done, it usually means something is not right. Most likely one of your sysadmins has updated a piece of software without syncing the children. Or worse, a sysadmin might have changed a child directly, instead of doing the change on the parent. If this happens a lot you'll need to have a little chat with that person.

But you could also encounter unexplained changes. This could be a sign of a security compromise. If you suddenly notice that in.telnetd, sshd and inetd.conf are going to be updated, you probably have a problem. [9] Running this command daily and checking the output is a good way to notice any changes to your systems and take appropriate action.

Reacting to security problems

New security holes are discovered on a regular bases. Quick reaction is often necessary to prevent yourself from being massively compromised. If you have hundreds of servers, you might be too late in updating them all.

Our system has allowed us to react to these types of problems very quickly. Once a hole is discovered, it typically takes us less than an hour to install and propagate the changes throughout our tree of servers. And because, in theory, the tree covers all your servers, you are certain that all your servers are indeed updated.

As a result, the amount of compromises has significantly decreased.

Upgrades

One of the most painful projects for anyone running any amount of servers is upgrading the OS. Even between minor versions of operating systems there usually are large differences. For this reason we keep different OS versions in different trees. Upgrading a server means switching it from one tree to the other. This sounds easier than it really is.

Upgrading the OS master can be done in several ways. You could use the OS update method. This is different for each OS. In our experience this often results in problems because we sometimes install binaries from source instead of using the systems own packages.

Going from FreeBSD 4.2 to 4.5, we decided to install a 4.5 OS master from scratch, but use the exact same disk layout as our 4.2 systems. We also installed all the packages that were installed on the 4.2 systems.

[9] We actually had this happen once. Due to a miscommunication several servers were installed outside of our PXE installation process and were not synced with a master. When we found out a few days later, we tried to sync them and noticed a lot of inetd run binaries had changed. Turned out this server had been compromised because of a known telnetd bug, which of course had been fixed in the masters. Syncing them solved the problem in minutes.

We’re now in the process of upgrading our clusters one by one, by syncing the TEST0 server with the 4.5 OS master. After the TEST0 master is synced we propagate the changes to the TEST1 server, and see if everything works as it should. When we’re happy with the whole process, we make the TEST0 server the new master, and update all the children from the master. At this point you have, hopefully successfully, upgraded your whole cluster to a new OS version.

And if anything goes wrong after all, you still have your original OS master, so you are in theory able to revert back to the previous OS version by re-syncing all the children with the old master.

After a period of a few weeks we assume everything went ok, and the old master will become the new TEST0 server.

It does require some work, but the results are worth it. Keeping your servers updated is very important. Not doing so could seriously influence your future flexibility. [10]

Syncing during PXE install

After a PXE installation, a new server needs to be immediately synced with a master server to bring it up to date. At first we manually synced the server, but luckily FreeBSD allows for a method to do this during the PXE install, fully automating the whole process!

This is done by creating your own FreeBSD package and installing it using install.cfg. The package system allows for a pre and post installation script. We can use these scripts to start a sync with a master server. This can be different for every type of master by changing the package name in the install.cfg (see chapter 2).

We add the following lines to install.cfg:

```
package=rsync-2.4.6
packageAdd
package=postdh-1.0
packageAdd
```

This installs rsync and a custom package on top of the base install. Building the custom package requires a few easy steps. First you’ll need the following files:

PLIST	this file can be empty
COMMENT	a short description of the package

[10] As an example, we ran into problems when we wanted to use specific gigabit Ethernet cards in our infamous newszilla Usenet cluster. The cards weren’t supported in FreeBSD 4.2.

DESCR	just use the same description as the comment
pre	a pre-install shell script. This can contain anything you like, but we keep this one empty.
post	this is the script in which we do the actual syncing.

You can then create a package out of this by putting these files in a directory and running the pkg_create command (see manual page). You can also use a small shell script made by Alfred Perlstein at http://people.freebsd.org/~alfred/pxe/pkgmaker.sh

The post script can be used to run the actual sync. As described before, you'll also need ssh keyfiles and an exclude file. Since we'll need to access these files during the install, we put copies of these files inside the FreeBSD distribution we have on our NFS server in a directory we call "files". A copy of our post-install script can be found at http://www.xs4all.nl/~scorpio/sane2002.

After adding this feature, we're now able to install and synchronize servers by just turning them on and plugging them in an Ethernet port. A few minutes later they're fully ready for production work in one of our clusters.

4. Problems

A solution wouldn't be a solution unless it created some new problems. For most people these problems will turn out to be minor.

The techies

The first problem we encountered was a strange one. Since installations are automatic, and maintenance follows relatively strict rules, some of the technical crew did not immediately like this setup. They felt that they were restricted in some ways, which is of course actually true. They suddenly couldn't just login a specific server to quickly fix a small problem. Instead they had to use the master server and sync the changes to the children, keeping in mind exclude files.

Also the strict adherence to specific operating systems and versions met with some resistance (and still does unfortunately). Everyone has his or her favorite OS, sometimes changing with the seasons. All kinds of arguments will be presented why one should change to this latest and greatest OS/version. But in the end, the benefits or often minor and most likely more influenced by personal tastes. You cannot fulfill the personal tastes of dozens of techies, without reverting back to the dark ages.

Luckily most of the techies now fully support our centralized system, because in the end it saves everyone time and money. We of course do sometimes install servers outside any tree. Specific needs and specific possibilities of an OS drive these choices. No system should be so strict that you can't deviate from it once in a while.

The sales department

Another, more serious problem, is still not fully solved. Since the early days of XS4ALL we went out of our way to accommodate the customer’s specific needs. Our sales department, usually (and hopefully) after consulting with the technical department, would sell customers individually tailored solutions. But as more and most clusters, and specifically the Dedicated Hosting clusters, got put into our centralized maintenance, these individual changes were increasingly creating problems.

Every individual change you make effects the complexity of your master/child pairs, making it more often then not necessary to make changes on production systems after all. Something we were really trying to avoid.

This created some tension between the sales and technical departments. We were telling them they could no longer sell individually tailored solutions, since this would severely increase the operating costs of the clusters, and therefore the cost of the products involved. They weren’t used to hearing that, and even less used to telling the customers the bad news.

As time goes on this issue is slowly getting resolved. Often the changes can be done in customer-data areas. And if not, most customers understand our reasoning for not fulfilling all their wishes. We have also created a new product line where personalized changes are possible, but for a higher price.

You will have to keep this field of tension in mind when going from a can-do environment to a more restricted one.

Extra hardware requirements

It is probably obvious that this system does call for extra hardware components. You will need master servers for all your clusters, and in some cases test servers. But since these servers can be low-end, the additional costs were, in our case, minimal. These costs are also investment costs, which can be written off in several years.

On the other hand it will save you a lot in operational costs. It will probably make your bean counters happy.

Restarting daemons

We have currently also not fully solved the problem of restarting daemons once they have been updated. If you update sshd or apache or any other daemon you will probably want to restart the daemons on all your servers that have been updated.

At the moment we do not try to do this automatically, which unfortunately does give us some manual work. But automatically restarting daemons opens up its own can of worms, where you probably will need to check if everything started up ok. We actually prefer doing this manually and controlled. Customers want their own daemons restarted when they're ready for it, which is often not at the same time we update the software.

For our own load balanced clusters, restarting daemons usually poses no problem because as soon as the daemon is down, the load balancers (we use Alteon hardware) will redistribute the demand.

5. Conclusion

It is now hard to imagine we ever managed to run our servers the way we did. It is even harder to imagine we managed to do it without any major operational problems. Had we continued our old ways, we would surely have run into a stonewall by now. You can only run so many servers individually.

As the ISP markets consolidate, pressure from management and shareholders to decrease operational costs will force technical people to run their systems more efficiently. But there is a fine, and sometimes dangerous line, between views of management and the views of technicians. Many ISPs and other large organizations have scared their technical departments away by forcing rigorous changes, forcing their techies to manage systems and software they are not accustomed to or just plain dislike.

Changing our server maintenance to this PXE/rsync based system has significantly decreased our operational costs. Not only in reduced personnel requirements, but also in more efficient use of resources. It has given both management and the techies what they wanted

It has dramatically affected the way we work, but without affecting the way we *like* to work. Servers are installed and made fully operational in minutes instead of hours or even days. Maintenance can be done by a very small group of people, freeing up everyone else for more important things like developing new products. Server failure has stopped being the problem it used to be, giving us more time to focus on scaling issues and other 'growing pains'.

Finally, and for most of the readers of this paper maybe most importantly....

We all sleep a lot better.

References

1. Booting FreeBSD with PXE. Much of the PXE install was based on his paper and used with his permission. http://matt.simerson.net/computing/freebsd.netboot.shtml
2. FreeBSD Jumpstart guide. http://people.freebsd.org/~alfred/pxe
3. Wired for Management. http://www.intel.com/labs/manage/wfm/wfmspecs.htm
4. Sysinstall, rsync, ssh, etc man pages.