

NETWORK EVERYTHING



COVER
FEATURE

From simple X connections to building multifunctional WiFi routers, there's no end of fun to be had with a network...

Networks are pervasive. No sooner are two devices networked together than someone will establish a need for a further half a dozen to join the throng. Any network is really about sharing information. It could simply be your desktop PC and a PDA sharing calendar information, or it could be sharing a complete system so that you can control one device remotely.

Of course, there are many ways to network different devices together, from the constraining but super-fast cable-based systems of Ethernet and USB to the wireless ways of Bluetooth and WiFi. But more important than how you connect devices together is what you can actually do with this newly found empowerment. Over the next 16 pages, that's exactly what we'll deal with, demystifying some of the technology as we go. The network is important, but what travels over the network is even more so...

CONTENTS

BLUETOOTH.....	PAGE 45
DOING IT WITHOUT WIRES	PAGE 46
NETWORK SERVICES	PAGE 48
SSH.....	PAGE 50
REMOTE X APPLICATIONS.....	PAGE 52
SECURITY	PAGE 54
REMOTE DESKTOPS.....	PAGE 56
DISTRIBUTED COMPILING.....	PAGE 58
PARALLEL COMPUTING.....	PAGE 59

AUTHORS

Andy Channelle, David Coulson, Richard Drummond, Graham Morrison, Roger Burton-West, Nick Veitch

BLUETOOTH

Virtually every tiny device seems to have a Bluetooth connection, but is it worth it?

Bluetooth has so far had an uninspiring existence. Originally designed as a more effective replacement to IrDA, the short-range, low-power wireless standard is

also low cost, meaning it can be included in lots of handheld devices.

Bluetooth itself is pretty flexible and can carry all sorts of protocols, which is where some of the problems with integrating Bluetooth devices start to occur. Many devices won't implement or be able to handle the full gamut of protocols or datatypes – after all, there's not much point in being able to connect your handsfree set with your printer.

However, some devices don't bother to implement all the systems to handle data that they should be able to accept. For instance, there are plenty of mobile phones – picture phones at that – which only implement a basic Bluetooth dialout connection, meaning that if you want to send images from your laptop, you still need to use the cable.

So in the confusing world of Bluetooth, the nascent Linux software isn't completely to blame when things go awry – your mileage may vary.

OBEX Push

The OBEX Push protocol was designed to make it easy to send files from one device to another. It even works, depending on what type of device you're trying to connect to.

On the Linux side, the KDE *kbtobexclient* is one of the more reliable pieces of software. This enables the browsing of connections and has a simple file navigator for the sending of files.

This is primarily a one-way communication – you can send a file, but there's no way of reading the remote files on a device.

This is one of the Bluetooth services that works the most reliably under Linux, but it can depend a lot on what kind of device you're attempting to connect with. Because Bluetooth is most commonly found in PDAs and phones, for some reason device designers seem to think that you would only ever want to send messages or virtual business cards in this way, so if you send your phone a picture, you may find that it doesn't know what to do with it.

Even Pocket PC devices, which you might expect to be a bit smarter, try to open everything you send them with *Pocket Outlook*.

OBEX Transfer

OBEX Transfer is a step up from the push method because it enables two-way communication and therefore the ability to browse the destination device (or at least its storage), and read, as well as write, data. On handheld devices, this usually enables file access to a particular folder. Most handheld devices treat this protocol just for file transfer, which means that they'll just

store whatever files are sent, rather than trying to make sense of them.

BlueChat

Most of the Bluetooth chat programs effectively just use a straightforward serial connection to transmit ASCII data. Various chat clients are available, often on simple devices because, to be honest, there isn't a lot to it. On Linux, *kbtserialchat* will do the job.

Why you actually need a text chat program on a device with a range of only a few feet is open to question...

Other stuff

There are plenty of other things to be done. The KDE Bluetooth Tools project is still pretty active, so check it out at <http://kde-bluetooth.sourceforge.net>. This contains a number of tools, including all the stuff mentioned so far and with plenty more are in the offing.

For audio, there's *kbmused*, a Linux implementation of a system originally designed to act as a remote control for media players. Another project of note is Bluetooth-alsa (<http://bluetooth-alsa.sourceforge.net>), which aims to create an alsa device for Bluetooth headsets, meaning you can stream audio from your computer to a headset.

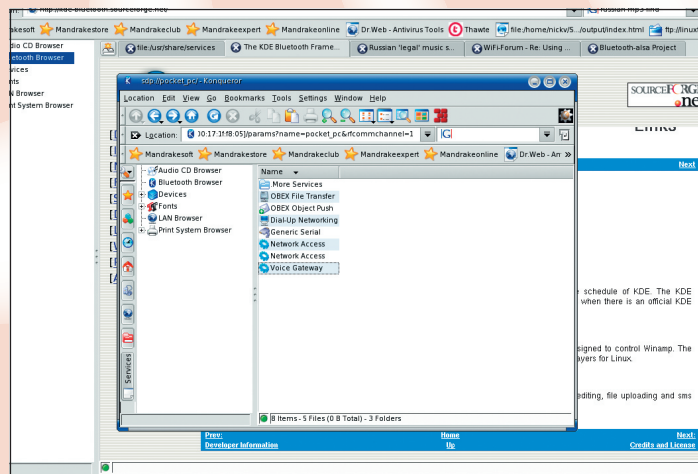
There's a slight problem with audio on Linux at the moment. The SCO protocol used for transmitting audio seems to fluctuate between working and not working, depending on which version of the kernel you're using.

SECURITY

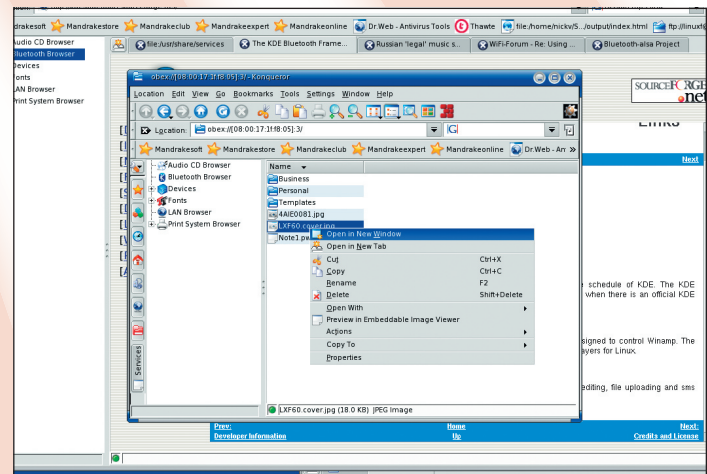
Bluetooth commonly uses a simple PIN system for security, and encrypts the data travelling between two devices. The PIN on many devices can be of any length or can be a password containing alphanumeric data. While this is more secure, many more simple devices opt for a four-digit pin, as specified as the minimum requirement.

It's worth checking the devices you intend to use beforehand because some of the more primitive ones also only allow you to enter a four-digit pin when trying to connect elsewhere, which may you some cause problems if you've set up a 27-character passphrase instead!

In any event, it's wise to implement some sort of security in case your device gets bluejacked.



The KDE Bluetooth software is well integrated into Konqueror, at least as far as browsing devices and services available.



The OBEX Transfer method is the most seamless way of browsing and transferring files between devices, if your hardware supports it.

DOING IT WITHOUT WIRES

Get to grips with wireless protocols and build your own router from everyday household items...

Wires are for squares, and going wireless can change the way you use a computer. It's just not possible, for instance, to carry your laptop down a flight of stairs to show off some photos if you're tethered to an unravelling cat5 cable. The use of wireless is fairly widespread now, and the good news for Linux is that most hardware is supported by stock kernels.

However, you can do more with wireless than just connect up your laptop to the Internet. Get to grips with the basics and you can turn that redundant old PC into a wireless router with bells, whistles and loads of other streaming media.

Pick a card

There's an old saying that says if standards weren't good, we wouldn't have so many of them. In the wireless space, these standards are formulated by the Institute of Electrical and

Electronics Engineers (IEEE). We're concerned with 802.11, the standard for local area networks (LANs).

802.11b is a protocol that provides up to 11Mbps (megabits per second) and operates in the 2.4GHz band. This is the standard WiFi product and is currently the most popular format for wireless connections. Products available include network cards in PCI, PC Card and USB form, plus factors, routers, bridges (to join wired and wireless networks), access points, hubs/switches and print servers.

Support for this protocol is now pretty mature in Linux, and as most cards contain chipsets from one of just a few vendors, there's little chance that whatever you purchase won't work. That said, I would recommend sticking with the more established vendors (such as Netgear, Actiontec and Belkin) simply because of critical mass. If you have a problem with some obscure generic PC Card, you may struggle to find help. By using an established brand, there's more chance that someone else in the Linux community may have encountered a similar issue and found a solution.

As with any hardware purchase, it's worth doing a quick Google search or combing through some hardware lists to find out whether others have had success (or not) with your potential purchase before you actually part with your cash.

802.11g theoretically offers 54Mbps in the same band as 802.11b. In addition to faster speeds, 54g, as it has been branded by some, is also capable of communicating over greater distances. A wide range of devices and cards is available for both desktop and notebook machines.

Recent kernels have added support for many 802.11g chipsets, but again, don't buy anything until you've checked out the usual sources of hardware information. However, with this more up-to-date hardware, you will find available data a little less

comprehensive. For instance, the Prism54 website, a space devoted to the drivers for cards based on the Prism series chipsets, designates as 'untested' a PCMCIA card (an Actiontec 54g PC Card) that LXF has had lying around for the best part of a year. We popped it into a laptop running Ubuntu Linux and it associated with a 54G access point, using network details already provided for an 802.11b card with no need for any configuration at all.

As with everything else, do your research before buying. Start at http://prism54.org/supported_cards.php. Look down the list, find



The D-Link G520 is probably the fastest WiFi card on the planet.

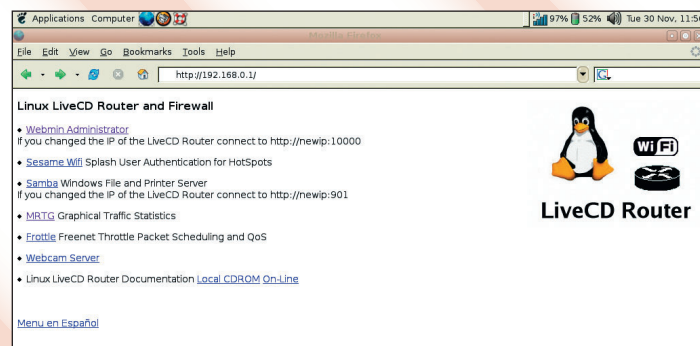
one designated 'Perfect' and start finding prices. Some of those that have achieved the Perfect rating include:

- D-Link Air Plus Extreme (PC Card)
- Netgear WG511 (PC Card)
- SMC EXConnect G (PCI)
- Z-Com XG900 (PCI)
- Linksys WRT54G (miniPCI)

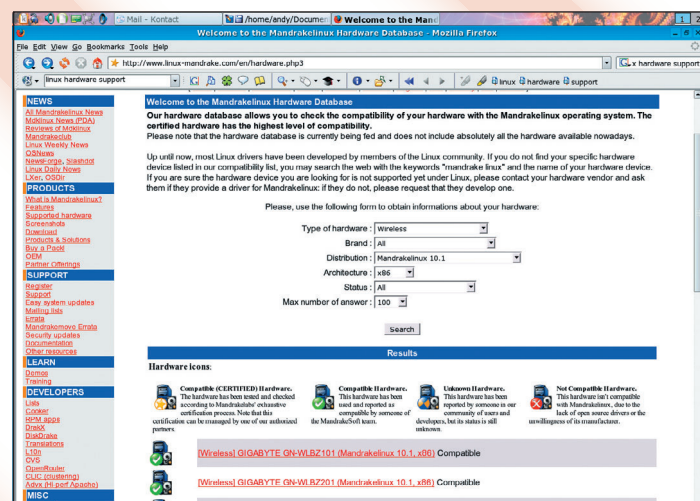
For unsupported cards, it's possible to use ndiswrapper (<http://ndiswrapper.sourceforge.net>), which allows for the use of native Windows XP drivers. It's a little more complicated to use and requires compiling against kernel sources, so, in terms of support, it's probably better to demonstrate to manufacturers that demand for native drivers exists.

A Linux router

The big advantage of running a PC-based router/access point with Linux is that it can do more than just one job. For example, our wireless expert's router doubles up as a media server – streaming MP3s and radio to



Remote access to LiveCD Router is basic unless you buy the Pro version.



Like most distros, Mandrake has a hardware guide.

PARTS OF A NETWORK

Networking jargon and terms explained

A network is more than just two or more computers. It's also defined by a number of steps in between that are either designed for simplifying management or for the purposes of security, and it helps, especially when it comes to troubleshooting problems, to know what's what. These options will all be configured in your distribution's Wireless-Tools frontend.

In Fedora, these are accessible via Menu > System Settings > Network; in SUSE, it's through YaST > Network Devices; in Mandrake it's in the Control Centre under Network; while in Ubuntu/Gnome distributions, you'll find the options under Computer > System Configuration > Network.

Firstly, the ESSID (or sometimes just SSID) is the Extended Service Set Identifier. In common parlance, that's simply the network name. As wireless networks become more and more popular, the ESSID is also becoming increasingly important.

For instance, within range of the author's house are a couple of networks that are called WORKGROUP – one of which is usually left 'open' with Windows file and printer sharing active – because the users haven't customised their setups. This identifying name can be a text string including up to 32 alphanumeric characters.

The Channel is, as you might expect, the frequency on which communications

take place. There are 11 available, which is useful if your network is in close range with other WLANs and there's a danger of crosstalk or interference.

It is also possible to configure the transmission rate, although it's usually best left at its highest setting. Occasions when you may need to hamper the speed of your network are those when you find you're getting an unusually high level of interference.

Sending data through the air, especially private data, is fraught with danger. Fortunately, the Wired Equivalent Protocol (WEP) can help secure communications from casual crackers. WEP requires the input of a string of characters before a client can connect

to the WLAN. These are either 64-bit, which requires a key of 10 hexadecimal digits, or 128-bit, which needs a 26-digit key. Remember that hexadecimal digits are the numbers 0-9 and the letters A-F. It may also be possible to use an ASCII key or a passphrase to generate the WEP key, depending on how your distribution is configured.

The sequel to WEP, not yet ratified by the IEEE but expected to be, is the WiFi Protected Access (WPA), which gets around the WEP problem of having static keys (especially in public networks and widespread client distribution) by using a Temporal Key Integrity Protocol (TKIP) in tandem with 802.1x technologies to provide dynamic key assignment.

computers around the house – and a testing web server. It can also, should the need arise, be pressed into service as an extra or emergency desktop.

It's possible to set up a router using a very basic, stripped, system. You don't need, for example, a hard disk, CD/DVD or sound system. In fact, if you use something like Freesco (www.freesco.org) you can set up a router, complete with firewall and a range of different servers, on a machine with as little as 8MB RAM and a 486 processor, though if you want to run without a hard disk, you'll need at least 17MB.

The machine we've set aside for this project has a CD drive, which is bootable, so we're opting for a more fully-featured product – in this case a Live CD firewall/router. There are a number of these available and we've chosen *Linux LiveCD Router*, available from www.wifi.com.ar/english/cdrouter.html. This is a live distribution based on Slackware and a custom 2.4.24 kernel. The ISO clocks in at 88MB.

You'll need to ensure that the PC designated for this job is CD-bootable. You can check this by hitting the DEL key at the start of the boot process to peek into the PC's BIOS settings. Look for an entry for the primary boot device, select this and change it to CD. If the BIOS doesn't support CD booting, the distro features a floppy disk image that can be used to bootstrap into the CD.

Linux LiveCD Router supports a range of WiFi cards, including those based on Prism2 or Lucent cards via the linux-wlan-ng and Orinoco drivers,

as well as dial-up support for a variety of modems with PPP.

Once the CD has been created, you can put it into the router machine and boot up. You can log in with the username 'root' and the password 'cdrouter'. The chances are that your network cards have been found automatically, but it's wise to do some manual configuration. Use:

```
netconfig eth0
```

to begin a console-based configuration tool for the network card. Simply follow the commands and enter the information (IP address, netmask and so on) as you would in any normal network setup. In this case, you would probably use DHCP for most options.

Next, do the same for the internal network card. If it's a wired card (connected to a wireless access point), it's simply a case of inserting the correct IP address (usually 192.168.0.1) and netmask (255.255.255.0).

Now set up the routing so requests from the internal network get sent out through the external connection:

```
ifconfig eth1 192.168.0.1
```

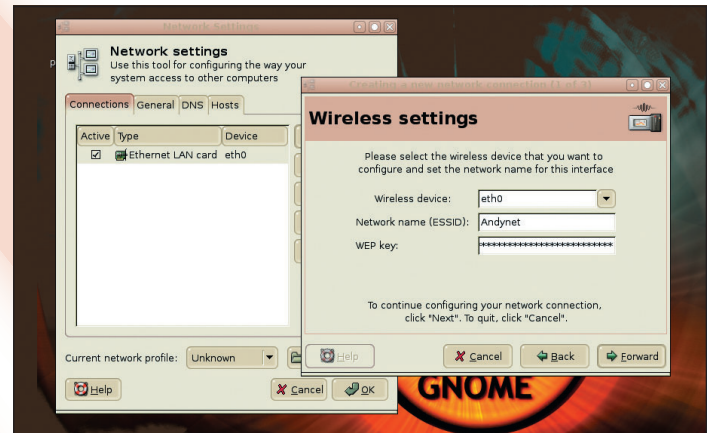
```
route add 192.168.0.1 eth1
```

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

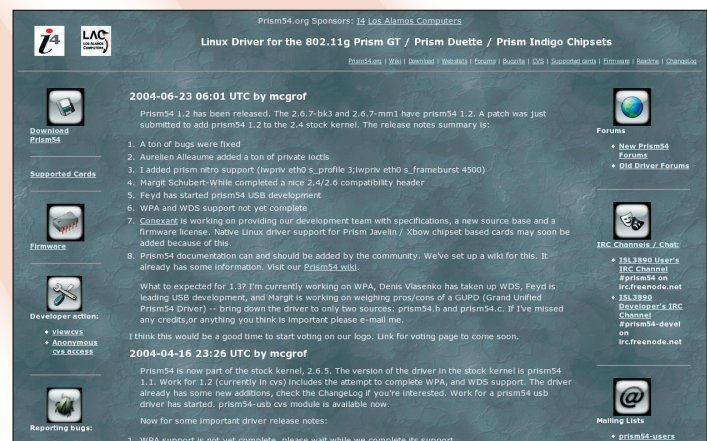
Finally, you can restart the network system to put the changes into effect:

```
/bin/sh /etc/rc.d/rc.inet1
```

The developers of *Linux LiveCD Router* claim that most wireless cards will be picked up automatically, and this was borne out by our (necessarily) limited testing. However, there are some options that need to be set for the smooth running of the WLAN. All of the configuration options are stored in `/etc/wlan/wlan.conf`.



Ubuntu's wireless GUI is quite simple. Well, it is the standard Gnome one!



The first job that's required is to set up the ESSID (we're calling ours 'LXFNet') by doing the following:

```
cp /etc/wlan/wlancfg-DEFAULT /etc/wlan/wlancfg-LXFNet
```

You then need to add this line:

```
edit /etc/wlan/wlan.conf
```

Add the name of the network (LXFNet) to the SSID_wlan0= line. Now edit `/etc/wlan/wlancfg-LXFNet` to add other network details, such as WEP keys and you're done!

The Prism 54 website should be your first stop before purchasing an 802.11g network card.

NETWORK SERVICES

Shared storage and printing facilities can save on resources and, ultimately, money.

Accessing devices on other systems over a network is probably the most common use for any home or office LAN and can often help to save a significant amount of money. Being able to share one printer across five or six machines, or store files on a shared storage device for easy access, quickly becomes second

nature to even the most blissfully ignorant of users. Also, not having to worry about users blowing away their workstations, because their important data is stored on a shared system well out of their reach, solves many problems.

This architecture does require a little behind the scenes configuration and administration but, once done, it can be left to run itself and doesn't need its

hand held, leaving time for important things, such as helping users figure out where My Computer is on the desktop.

For Unix systems, the standard method for sharing storage is through the Network Filesystem (NFS), which for many years was considered the most unstable piece of software in existence. However, with the current NFSv3 and NFSv4 implementations, the reliability of NFS has increased significantly, as has its ability to recover from network failure, making it usable on a variety of environments.

NFS, because of its implementation and the need to access certain resources, is particularly insecure – it isn't something we would ever want to dream of running in plain IP over the Internet. On a LAN, or over a VPN, it does make for easy access to other systems, particularly as it doesn't care at all what we're connecting to, or what we're connecting from. We can NFS mount a directory containing a mounted CD-ROM over to a Solaris box, or share the /home directory on our fileserver across multiple workstations without a great amount of reconfiguration.

One thing to remember when using NFS is that, because it's a Unix filesystem, it's heavily dependent on the UID/GID structure across the server and client being the same. If we have the UID 500 on one box, and UID 1000 on the other, don't expect to be able to write to files without making some changes to /etc/passwd first.

While this may seem annoying, it does fit with the structure of NFS when used to share home directories on Unix boxes where the UID of each user would be the same on each.

Configuring an NFS server is as simple as editing one file because all we need to do is add the appropriate directory and permissions to /etc/exports. Within the permissions, we can set specific rights to individual IP addresses or network blocks so that, for example, if a general workstation mounts the filesystem, the root user on it can't access or modify files on the NFS mount as if it were the root user on the server.

Securing the NFS mounts is a particularly important part of building an NFS server, and taking some time to understand all of the /etc/exports

options before throwing it out into the world is a good start. As always, 'man 5 exports' provides detailed information on all of the options.

If we wanted to share /home on our local network, we would add the following line to our /etc/exports:

```
/home 192.168.1.0/255.255.255.0
(no_root_squash,rw,sync)
```

Note that there should be no spaces between the netmask or IP and the first parenthesis, otherwise we will give default mount access to 192.168.1.0/24 and allow the whole world to mount it with r/w permissions. This is, of course, somewhat insecure, and should be avoided at all costs.

Once /etc/exports is modified, we can refresh our NFS server with the changes, known as 'reexporting', using:

```
# exportfs -r
```

We can test our current export list using the following, to ensure that our changes are active:

```
# exportfs -v
/home 192.168.1.0/255.255.255.0(rw,wdelay,root_squash)
```

We can now head over to our client and attempt to mount the /home directory from the server, using the standard mount command.

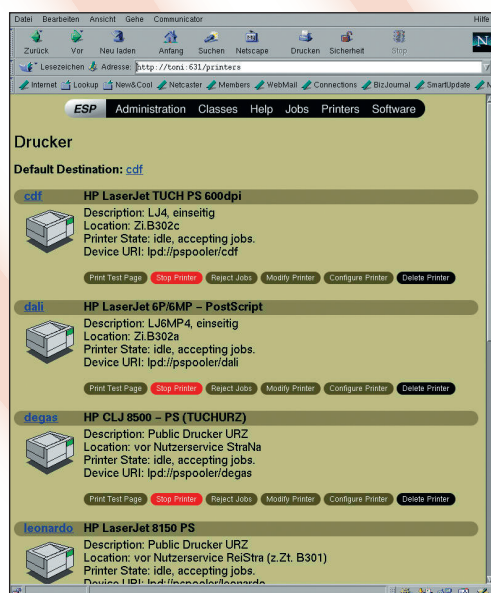
Assuming our server has the IP 192.168.1.2, we would do:

```
# mkdir /home-nfs
# mount 192.168.1.2:/home /home-nfs
```

This will mount /home from 192.168.1.2 onto /home-nfs on the local system. We can also add NFS mounts to our /etc/fstab file to have them mount at boot time:

```
192.168.1.2:/home /home-nfs nfs
defaults 0 0
```

To share a directory between a Linux server and a Windows box requires a little more imagination because the popular *Samba* package is required. *Samba* provides all of the Windows file and print sharing capabilities to a Linux system, enabling us to share a Linux filesystem on a Windows box, mount a Windows share under Linux, or access a printer connected to a Linux system from Windows. Any distribution that comes with more than 'cat' will have *Samba*, so if it isn't installed already, it will be



CUPS has a web-based administration tool, avoiding the need to hack at text files manually to make the printer work.

FTP SERVICES

For basic file sharing, nothing beats FTP for public or private access.

While numerous methods for accessing remote files have popped up over the last decade, including *Samba* and *WebDAV*, good old FTP is still the most popular way to get a file from A to B over a network with the least amount of hassle. Nearly all Linux software is distributed via FTP, and it's not uncommon for vendors and support companies to provide FTP services to distribute software to users.

Building an FTP server is easy, and there's quite a choice of FTP software available. *ProFTPD* and *wu-ftpd* are popular. The former has a wider feature set, including authentication through SQL or LDAP, as well as virtual hosting. *wu-ftpd*, on the other hand, supports basic authentication using the local /etc/passwd system, which is more than enough for general FTP installations. Once authenticated – or connected as

'anonymous', which will often map to 'ftp' – the FTP client will see the home directory for the user and have the appropriate download and upload permissions as defined by the directory.

FTP uses Plain Text authentication, so user authentication should be avoided over a public network where the passwords can provide access to internal services. Sniffing passwords from a switch or router is easy, and there's no standard, secure FTP system. Both *sftp*, provided by the ever-popular SSH package, and *ftp-ssl* do encryption and authentication in different ways, and some clients support *sftp* and some support *ftp-ssl*.

If security is a concern, using *sftp* and avoiding FTP altogether is a good option, and clients will have no choice but to use an *sftp*-capable client to connect to their FTP services.

Printing using *Samba* is very simple, because all we need to do is define a 'printers' section in `smb.conf` and *Samba* will share out whatever printers it can find. When we access these through Windows, the specific

CUPS

Printing with Linux is easy when using CUPS for spooling and processing

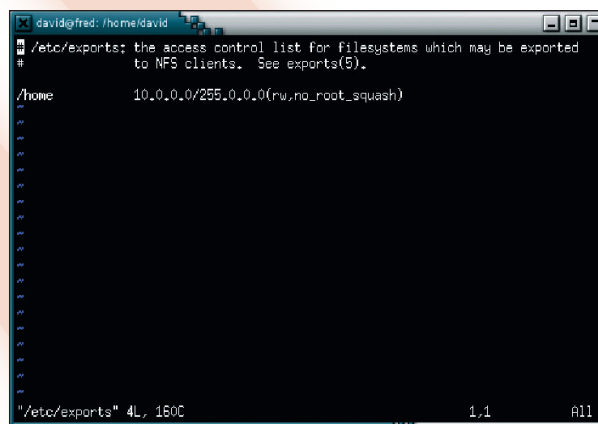
For many years, lpr was the standard printing process used by Linux systems and was installed by default by the majority of distributions. While lprng is faster, bigger and better than lpr, a completely different print management system has taken over the Unix print world. CUPS provides all the capabilities traditionally delivered via lpr yet with lots of additional functionality, making it far more useful on modern Linux boxes.

CUPS can be managed through the web interface on <http://localhost:631> or by using one of the many GUI tools that

exist for desktops, such as KDE or Gnome. None of the file hacking, or creating shell scripts to pipe Postscript through *GhostScript*, is necessary with CUPS, because it will do everything we need smoothly and magically. Well, almost magically – we still have to tell it which printer filter to use and set up the various spools for printing. However, as CUPS support the Internet Printing Protocol (IPP), it will pretty much work with any printer because most vendors support IPP now that it's in Windows. IPP also allows the print spool to be

reviewed over a web browser, avoiding the need for users to question where their print job went, even though they sent it to the printer half an hour ago.

Nearly every distribution installs with CUPS as default, so the chances are that if you have a printer under Linux, you're already using CUPS. However, anyone who battled with lpd back in the old days, and spent hours making their printers work smoothly using scripts, will be sickened by the ease with which CUPS allows administrators to manage the printing resources on their network.

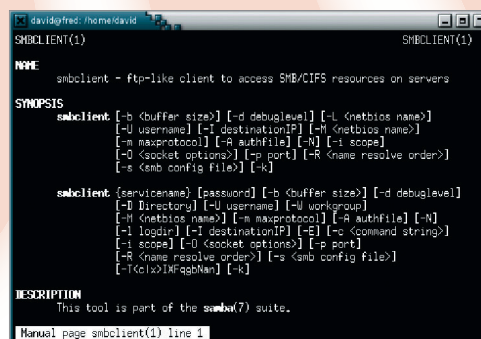


/etc/exports contains the directories we choose to permit NFS clients to mount, as well as a number of more cryptic options.

Windows printer driver will be required on the client, so no further configuration is needed. The printers section would look like this:

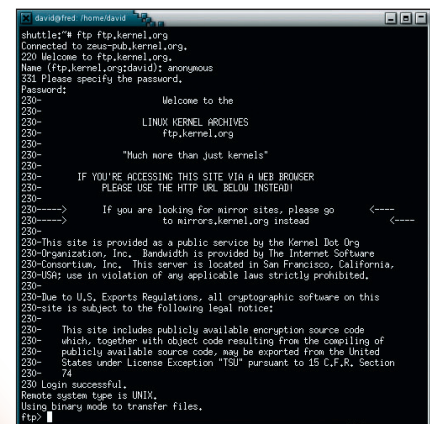
```
[printers]
comment = All Printers
browseable = no
path = /tmp
printable = yes
public = no
writable = no
create mode = 0700
```

Samba is a very complex package and provides capabilities well beyond the needs of most users. It effectively bridges the gap between Windows and Linux systems and can even provide some network capabilities that are



unavailable under Windows. However, Microsoft and the *Samba* team are currently playing a fairly humorous game of trying to see who owns the NetBIOS protocol...

smbclient is comparable to an **FTP** client. However, we can access **Windows** shares from **Linux** devices, enabling both file transfers and printing.



FTP is used to distribute nearly all Open Source software, including the Linux kernel.



The *Samba* project enables Windows and Linux to share storage and printers, without any additional software being installed on Windows systems.



SSH: THE SECURE SHELL

Telnet is dead. For a more secure environment, you should be using SSH. It's easier, too!

SSH (Secure SHell) is a replacement for the Telnet and rsh systems commonly found in older Unix installations. It also goes further by providing solid cryptographic protection to all data, allows other protocols to be tunnelled through its encrypted channel, and it supports passwordless logins too.

There are two parts to the SSH system: a client program, also known as SSH, and a server, sshd, to which the client connects, normally on port 22. There are also two versions of the protocol. Version 1 is rather less CPU-intensive than version 2, but cannot be considered secure in some cases, so

don't use it unless you have a very old machine that can't handle version 2.

Setting up keys

One of the biggest advantages of SSH is key-based login. This uses public-key cryptography to allow the user to log in to remote machines with a key (stored on the local machine, or even on an add-on pen drive or key fob) rather than a password. The key is kept encrypted; the user enters the passphrase once, when he logs in, and all other logins can then be done automatically. As always with public-key cryptography, you generate a pair of keys – the secret key, which you keep to yourself, and the public key, which you distribute to the machines you want to log into.

Generate a keypair on your desktop machine with this code:

```
$ ssh-keygen -t dsa -C your.email@address
```

Enter a passphrase for your key so that it can't be used by anyone else with access to the machine. For advanced use, you can set up a key without a password.

This is best kept to machines that are on secure networks, although it can also be useful for automatic file transfers.

This procedure puts the secret key in `~/.ssh/id_dsa` and the public key

in `~/.ssh/id_dsa.pub`. Now see if you have an ssh-agent running:

```
$ echo $SSH_AGENT_PID
```

Most window managers will run it automatically if it's installed. If not, start one up with this code:

```
$ eval $(ssh-agent)
```

Now tell the agent about your key:

```
$ ssh-add
```

and enter your passphrase. You'll need to do this each time you log in. If you're using X, try adding:

```
SSH_ASKPASS=ssh-askpass ssh-add
```

to your `.xsession` file to get prompted for the key's passphrase each time you log in. You may need to install the `ssh-askpass` program as well.

Now for each server you want to log in to, create the directory `~/.ssh` (for your username on that server) and copy the file `~/.ssh/id_dsa.pub` into it as `~/.ssh/authorized_keys`. Once that's done, you should be able simply to enter the following line:

```
ssh username@remote-box
```

and all the authentication will be handled automatically.

If you started the ssh-agent by hand, kill it with `ssh-agent -k` when you log out (for example, in the `~/.bash_logout` script).

Configuration

Most of the configuration of an SSH client happens on the command line – see the SSH man page for a full list of options, but bear in mind that the defaults will work almost all the time. One exception, though, is the choice of compression. This adds latency – even on a fast machine, it takes more CPU cycles – but saves bandwidth. It's particularly useful if you're transferring easily-compressed data, such as log files. You should use the `-C` flag to the SSH command.

SSH host configuration is done in the `sshd_config` file (typically in `/etc/ssh/`). The installed default is usually reasonable, though you may want to set `PermitRootLogin` according to your root access policy, and possibly turn on `X11Forwarding`.

You can also add special options to the `authorized_keys` file to allow some keys only to run specific commands. This isn't as flexible as might be



LOCKING OUT PASSWORDS

Block out password crackers with these simple changes

Many server operators have noticed a rise in SSH brute-force password cracking attempts in recent months. A password is intrinsically less secure in the face of brute-force searching than a key, since it's usually up to 16 typeable characters (about 53 bits), as opposed to the 1024 bits or more of an RSA or DSA key. To tell the server not to allow password logins and only accept keyed logins for root, set the rather counter-intuitive:

PermitRootLogin without-password in the `sshd_config` file. To turn it off for every account, set:

PasswordAuthentication no in the same file.

```
roger@parhelion:~$ ssh-keygen -t dsa -C roger@example.com
Generating public/private dsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/roger/.ssh/id_dsa.
Your public key has been saved in /home/roger/.ssh/id_dsa.pub.
The key fingerprint is:
17:c9:f9:a3:84:56:0b:65:1d:2d:62:3c:a5:b7:0c:5d roger@example.com
roger@parhelion:~$
```

Key generation in action.

```
File Edit Search Preferences Shell Macro Windows Help
# Package generated configuration file
# See the sshd(8) manpage for details

# What ports, IPs and protocols we listen for
Port 22
# Use these options to restrict which interfaces/protocols sshd will bind to
#ListenAddress ::
#ListenAddress 0.0.0.0
Protocol 2
# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
#Privilege Separation is turned on for security
UsePrivilegeSeparation yes

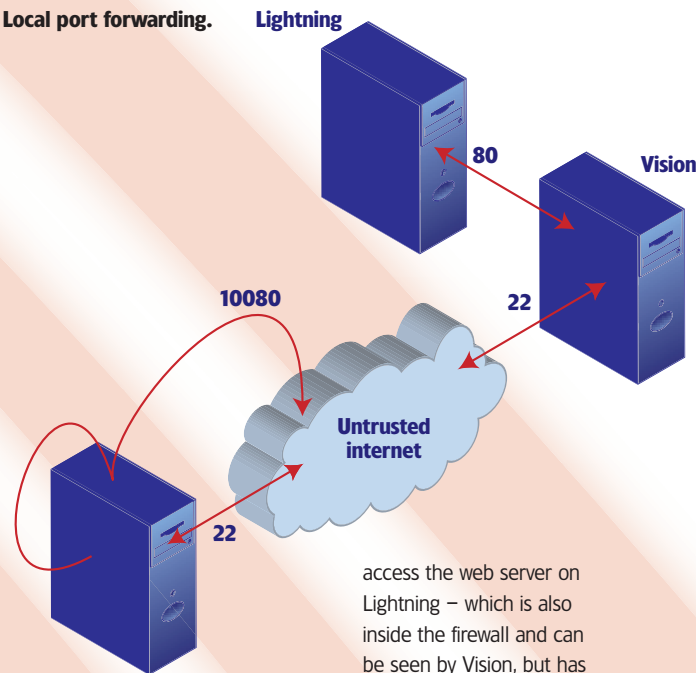
# ...but breaks Pam auth via kbdint, so we have to turn it off
# Use PAM authentication via keyboard-interactive so PAM modules can
# properly interface with the user (off due to PrivSep)
PAMAuthenticationViaKbdint no
# Lifetime and size of ephemeral version 1 server key
KeyRegenerationInterval 3600
ServerKeyBits 768

# Logging
SyslogFacility AUTH
LogLevel INFO

# Authentication:
LoginGraceTime 600
PermitRootLogin yes
StrictModes yes
```

The `sshd_config` file: it looks more complicated than it is.

Local port forwarding.



wished, but it's handy for automated system maintenance setups. See the `sshd` man page for more details.

Port forwarding

One of the most powerful features of SSH, and also one of the most misunderstood, is its port-forwarding (also known as 'tunnelling') ability. This enables you to push any TCP connection (not UDP) through the encrypted SSH connection to a remote machine. There are two different modes in which this can work:

(1) Local port-forwarding. This forwards a local port to a specific machine and port on the remote end. For example, say you pick up your mail by POP3 (from a box on which you also have an SSH login) but would like to encrypt the connection. Easy: just forward your port 10110 (or any other port you're not using) to port 110 on the remote machine, then point your POP3 client at localhost, port 10110.

```
ssh -L 10110:localhost:110
username@remote-box
```

The first part of the argument to the `-L` parameter is the local port. The second is the machine name (in the context in which the remote machine will see it – so you can forward to machines inside a private LAN, as long as you can get access to one machine to SSH into in the first place). The third is the port on that remote machine.

As a more complex example, you have an account on Vision, which is inside a restrictive firewall but allows SSH connections in. You want to

access the web server on Lightning – which is also inside the firewall and can be seen by Vision, but has no outside access – and to let other people on your LAN see it, too. Just use:

```
ssh -g -L 10080:
lightning:80
username@vision
```

People pointing their web browsers at port 10080 of your machine will get transparent access to Lightning's web server.

Note that SSH will run in the foreground as usual, and give you a command prompt. Closing this will close all the tunnels as well.

(2) Remote port-forwarding. This is essentially the opposite of local port-forwarding – you use it to expose a port on a local machine to the outside world. This is rarely useful, but can be handy for low-cost and low-effort VPN setups, where you want to expose a mail server or something similar to a

ROOT OR USER?

You need root access. There are two ways to get it.

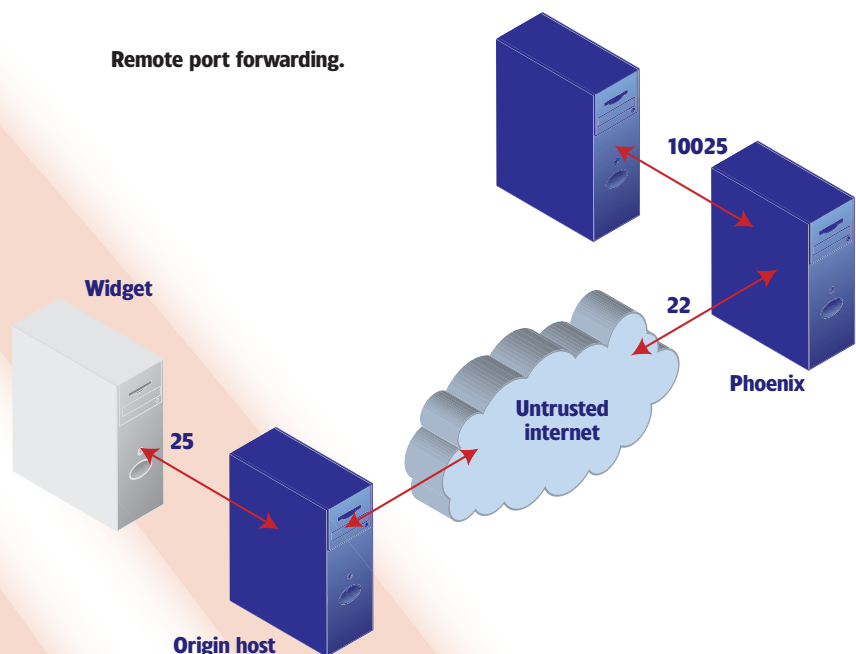
If you're using SSH for remote administration, there are two ways to get root access: either log in directly as root, or log in as a user and then use `su` or `sudo` to get root privileges.

Both methods have their advantages. Direct root login makes it easier to automate tasks with scripts and removes the need to set up accounts for every administrator on every server in the

network. Using `su` leaves a clearer audit trail (though it is of little help against a malicious insider), and encourages administrators to use root privileges only when necessary.

Pick the solution that seems best for your network. If you do want to disable root logins completely, set `PermitRootLogin no` in the `sshd_config` file.

Remote port forwarding.



group of machines on the other side of an untrusted network. Let's say the SMTP server is Widget and the host at the other end is Phoenix. Use:

```
ssh -R 10025:widget:25
username@phoenix
```

Users in the network with Phoenix then use port 10025 as their SMTP host. If you log in as root, you can even make this port 25.

X-FORWARDING

Run graphical programs on remote machines from the convenience of your desktop

This is really a special case of local port-forwarding, but since the internals of X are somewhat complex to set up, it has its own configuration. The server's `sshd_config` needs to have

```
X11Forwarding yes
X11DisplayOffset 10
```

(or some other number greater than the number of concurrent conventional X servers; 10 is almost always sufficient). Then, on the client, run:

```
ssh -X username@remote-box
```

You should then be able to run graphical programs on the remote

machine, and have them appearing on your X desktop just as if you'd run them locally. Note that this isn't as bandwidth-efficient as a protocol such as VNC, which was explicitly designed for congested links, but it can be very useful across a LAN. It's also possible to use it securely over the Internet if you're prepared to put up with some delays.

Note that since X was designed before the concept of malicious code became popular, you shouldn't use X-forwarding, other than to a box you trust. Suitable software on the remote machine could

request any information available to your local X-server. This is just as much a risk as with `xauth`.

One specialised use of this is to run X programs as root on your own machine. This is mainly for simplicity. It's easier to `ssh -X root@localhost` particularly if you've set up a key to allow this to happen automatically, than to `su` to root and then run `xauth -f /home/user/.Xauthority extract - :0.0 | xauth merge -`

```
export DISPLAY=:0.0
```


REMOTE X APPLICATIONS

Sometimes there's no other way and you just have to run desktop applications from a remote client. Enter X...

The X Windowing System (commonly known as X) appears to offer roughly the same functionality as other graphical platforms, such as Microsoft's Windows. It displays windows for a start, and you can move a cursor around the screen. What may not be immediately obvious is that X is quite a different approach from simply rendering graphics on a display, and the visual frontend hides a wealth of functionality just under the surface.

From the start, X was developed to be a platform-independent windowing system with the intent of making computing resources easily accessible across a network of students. The implementation of X that most Linux users are familiar with is XFree86, although recently most distributions have switched to using X.org X11 Server, which is a fork of XFree86 taken before changes to the licence made it incompatible with most GPL-adherent distributions.

It's only a small conceptual step from platform independence to network transparency, and that's

exactly what distinguishes X from other graphical platforms. X is built around a client-server architecture, which may seem odd at first, but makes more sense when you make a distinction between the hardware and the software that drives it. The X server is responsible for managing the display hardware and input devices, while the graphical rendering and interaction is handled by what are called 'X clients'.

With this simple abstraction, it's easy to see how X clients can connect to an X server across a network, although the actual mechanics of the process aren't quite so simple.

Master and servant

The client communicates with the server through a protocol called X wire. This is a two-way, error-free byte stream that can be implemented in many different ways. With XFree86 and X.org the X wire protocol can be found in the shared xlib library, which has the ability to communicate across TCP/IP among other protocols.

The protocol is therefore device-independent and allows the client to

control basic geometric and textual rendering without consideration for the hardware involved or the distance between the client and the server. This means that a local connection would simply take advantage of using Unix sockets, while you could just as easily use TCP/IP for remote access.

One of the downsides with this approach to network transparency is its inherent bandwidth requirements. The server is responsible for updating all of the graphics that appear within a window. That means that if a window is obscured by another, the server needs to continually update the display (through the network), which, on anything other than a local LAN, will become extremely unresponsive.

For example, moving a window over a remote xclock running on a local display generates a continual stream of information as the local client communicates with the remote server, as well as peaks of traffic when parts of the window are occluded. Hopefully, with the recent addition of XDamage to X.org's X11 6.8 release, this inefficiency will become redundant.

The first thing to bear in mind with remote X applications is security. It's too easy to forget that all your keyboard and mouse movements are being sent over what is basically an insecure protocol. It's therefore inadvisable to use X remotely without third-party security unless you are working on a trusted network. There are, however, two security procedures built into the remote X system that can make it more difficult for any malicious user to break into a remote session.

Simple security

The simplest method of security is configured through the xhost command, which allows you to grant and deny access on a host name basis only. For example, on the local machine you could use:

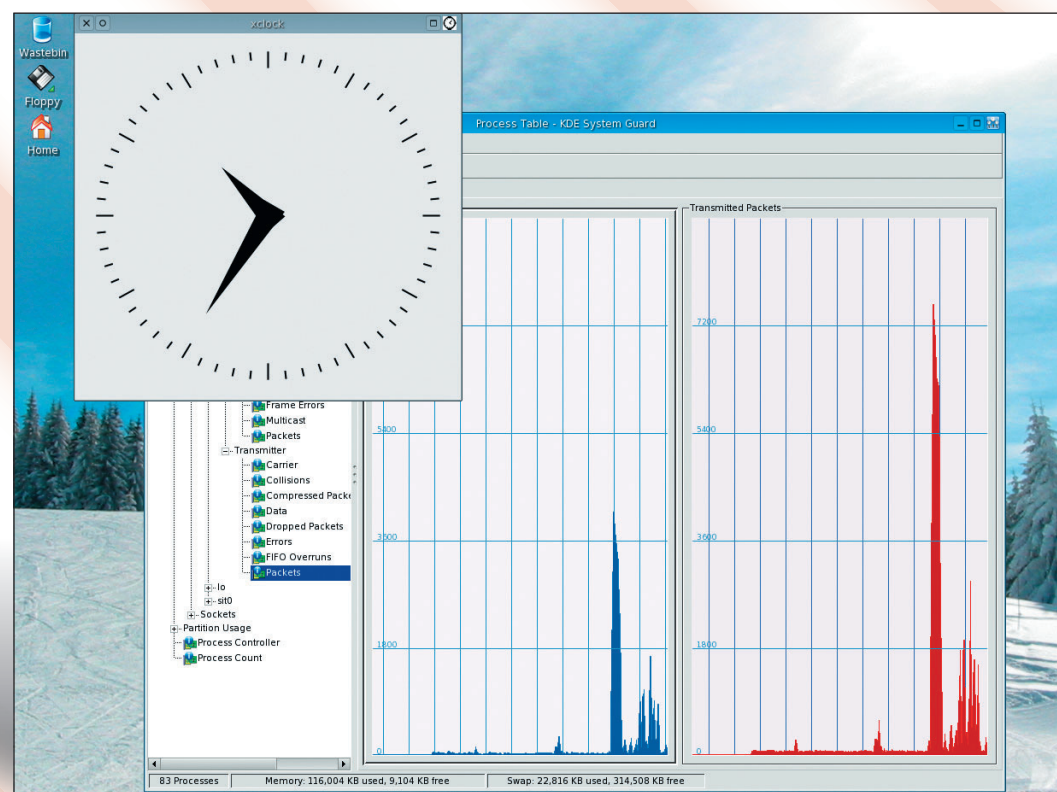
```
$ xhost +remote_client
```

This will allow connections from the remote_client host.

```
$ xhost -remote_client
```

This can be used to deny access to the remote_client, something that can

Resizing a remote xclock generates thousands of packets a second.



be safely done after you've opened your remote display to narrow the window of opportunity. Removing the host name entirely will either allow all connections or deny them, depending on whether you use a + or a -.

This may not seem like the most secure method of protecting your system from unauthorised access, and it's not. Not only does it allow any user on the destination_client domain to make a connection, but it also opens the door wide open to domain name spoofing attempts. The only advantages to using xhost are that it's easy to configure and widely available.

As the problems associated with xhost became all too apparent, it was obvious that a better method was needed, based around individual access. As you'd normally only need to grant access for yourself rather than all users of a system, this new system was built on personal authentication.

Authentication

Xauth is built around a system of sharing a secret passkey called an authorisation record. This is nowhere near as secure as a public and private key system, but it's still a lot more secure than xhost. The authorisation record is called MIT-MAGIC-COOKIE-1 reflecting the original developers from the MIT/X Consortium.

The first thing you need to do to get xauth to work is set the environment variable DISPLAY on the remote host. This needs to reflect the destination display you want the remote X applications to appear on. For the first display on the host machine, use *bash* and enter:

```
$ export DISPLAY=remote_host:0
```

Other shells treat environmental variables differently, of course. Authorisation records, or cookies, are randomly generated by xdm (or equivalent) when the X server is started, and reside in the .Xauthority file. You can list the cookies currently stored by using xauth list:

```
$ xauth list
```

```
remote_host/unix:0 MIT-MAGIC-
COOKIE-1 b077fade7e4d63fe08d8
0bc09ec1f5e0
local_client/unix:0 MIT-MAGIC-
COOKIE-1 40c2cb38b12e54e06d4
721bb488609c2
```

The first column of the output lists the display names as they would appear in the DISPLAY environment variable; this is composed of the inet

domain followed by the UNIX domain. The inet part is the one needed when setting the DISPLAY variable. The second column is the type of authentication scheme that needs to be used, and the third column is the 'secret' cookie itself.

This cookie is required on the remote_host to enable a two-way connection and also to prohibit programs that aren't directly controlled by you getting access to your display. If for some reason xdm didn't create the cookie, it can be generated using the following command:

```
$ mcookie sed -e 's/^/add :0 .
/'xauth -q
```

```
xauth: creating new authority file /
home/user/.Xauthority
```

The next stage is to transfer the contents of the cookie to the destination host. You could copy the key to the remote machine using something like *rsh*, or enter the key directly at the command prompt. The following command adds the key from the local_client, as used earlier:

```
$ xauth add local_client:0 MIT-
MAGIC-COOKIE-1 40c2cb38b12e5
4e06d4721bb488609c2
```

If all has gone well, you should now be able to execute X applications on the remote host that automatically authenticate themselves and open on your local display. Despite being significantly more secure than xhost, xauth still has a couple of obvious vulnerabilities. The authorisation records are transmitted over the network without encryption, making it theoretically possible for the cookies to be intercepted and used to make unauthorized connection. The cookie can also be 'guessed' if the key is created using a poor pseudo-random number generator.

It's also possible to run an entire remote desktop on your local machine. This is a good way of making the most out of older hardware that doesn't perhaps have the power to run things such as KDE or Mozilla directly. The remote desktop is configured through the display manager, which is the graphical interface that asks for a username and password when the machine boots. Traditionally, this program was xdm, but on recent systems it's been replaced with either kdm for KDE or gdm for Gnome.

Xdm manages the connections through XDMCP, which is the display manager control protocol. With xdm

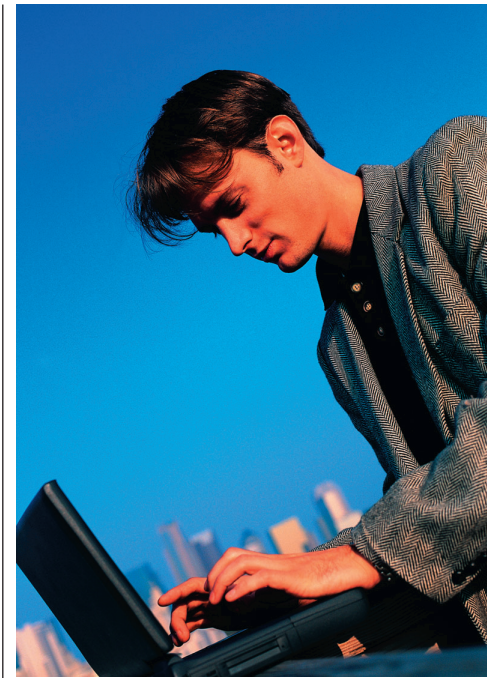
being responsible for generating the same authorization keys that xauth uses, you get the same high level of security. The configuration files for xdm are usually located in /etc/X11/xdm with corresponding locations for kdm and gdm. In terms of configuration, a couple of files need to be changed to allow remote access. The request port needs to be commented out in xdm-config, and the remote system needs to be added to Xaccess to enable remote XDMCP queries.

The most obvious candidate for strengthening remote X security is SSH, which can tunnel the entire remote X protocol through its own secure encryption. On commencing a connection with SSH, the remote sshd server automatically generates the environment variable DISPLAY as well as configuring the xauth authorisation cookies to allow access to the SSH tunnel. Once created, the DISPLAY variable on the remote machine should read something like DISPLAY=localhost:10.0, which represents a safe value placed after the usual array of virtual consoles. As this is done automatically, interfering with the preconfigured settings can compromise security.

If you're using entire X sessions managed with xdm, you can't easily configure the same level of security that you get with SSH and remote X application. This is because XDMCP is using UDP port 117 as opposed to X using TCP port 6000, and UDP port forwarding is not currently supported with either SSH1 or SSH2.

There are a couple of additions to using SSH that can improve efficiency. Choosing a low-overhead cipher reduces the CPU usage without adding too much risk. Secondly, using SSH's built-in compression can reduce the significant demands that are placed on the bandwidth from the resource-hungry X protocol.

In the end, using a bandwidth-hungry protocol may not seem like the best way of running remote applications. This is especially true when you can use other protocols, such as VNC, that are especially designed for use across a network with constrained bandwidth. However, it can be effective as a quick fix, especially across a trusted network, and it certainly works more transparently than running a whole remote desktop on a local machine.



Give your friends access to a remote desktop.



SECURITY

If you really are going to network everything, you need to secure it all too!

Because Linux developed at around the same time as the Internet became popular, the kernel supports a wide range of infrastructure capabilities. Since the 2.0 series of kernels, Linux has supported IP packet filtering, more commonly known as a firewall, without the need for third-party tools or software. This makes Linux a great alternative to an expensive, and often restrictive, commercial firewall product from Cisco or CheckPoint, and in true Linux style, there are numerous patches and tools available for all kinds of options. Of course, crazy patches often come with stability warnings, and changes not in the main kernel tree probably aren't in there for a good reason.

Building a Linux firewall from scratch is extremely simple, particularly for anyone who is comfortable with

the command line and using console-based text editors. A router or firewall is no place for X, so manually changing the configuration is the only way to go in this world. Most distributions have basic configuration tools that work from a terminal, although many of the modern and friendly distros lack a nice way to change things from the command line.

Picking a distribution

Debian is a great distribution for a firewall, because doing a basic install uses less than 500MB of disk space, and it's extremely simple to limit the software installed to the bare minimum for security purposes. The most recent installer for Debian supports every NIC and storage controller you can shake a stick at, as well as being able to do a fresh install on to a software RAID-1 array. Software RAID, if a hardware controller is unavailable or outside of the budget, is a must on a firewall, because no one wants to spend hours rebuilding a box from scratch if a disk fails. With a pair of cheap disks, a lot of time can be saved using Software RAID to maintain a mirrored copy of the firewall installation on both disks.

Firewall basics

In both 2.4 and 2.6, firewalling is provided by the iptables tool, which hooks into the networking stack on the kernel using Netfilter. Often, the terms iptables and Netfilter are used interchangeably, but they each have a specific use and function within the Linux kernel. Netfilter is the system in the kernel that allows packet filtering kernel modules to hook into the layers

of the stack without having to modify massive amounts of kernel code each time. Netfilter modules exist for both ipfwadm and ipchains, the firewalling systems used on 2.0 and 2.2 kernels, as well as for iptables.

iptables is a combination of kernel modules, to provide the filtering and NAT capabilities, as well as user-space tools for configuration and management. Nearly every distribution comes complete with iptables support, so there is no need to recompile the kernel manually. However, there are lots of patches for iptables using the 'patch-o-matic-ng' package found at www.iptables.org, which some may want to compile into their kernel. In most cases these are unnecessary, but for more complex configurations with a wide range of applications and services, the additional modules often come in very useful.

As iptables is a user-space tool, it's provided as a package by distributions, and again, most of the time this is installed as part of a basic distribution install. We can check iptables is available and everything is working by running iptables -nL from the command line, which will list all of the rules presently loaded into the filtering part of our firewall:

```
# iptables -nL
Chain INPUT (policy ACCEPT)
target    prot opt source destination
Chain FORWARD (policy ACCEPT)
target    prot opt source destination
Chain OUTPUT (policy ACCEPT)
target    prot opt source destination
```

CHROOT JAILS

When you have no choice but to run insecure services.

As much as we may long for a perfect world, free from buffer overflows and other careless coding techniques, running services that represent a potential security risk against a network are a part of life. To be a completely self-sufficient environment, without depending upon an outside ISP for services, more often than not requires the use of a service that can open the network up to an attack. However, these services can often be run on a network within a 'jail', so if they are exploited or compromised, there is little anyone can actually do with the system.

Jailing processes in Linux is carried out with a tool called chroot which changes the root filesystem to a sub-

directory of our real server. Essentially, this means we have two separate Linux installations – one being the complete distribution with everything we need and the other being the bare minimum to run a binary, such as 'named' from BIND. The second root directory generally only contains the required configuration files and libraries, and even in the latter case, it's usually easier to simply statically compile a binary so it doesn't require any separate libraries to be loaded.

Should anyone break into the system through the chrooted service, all they can do is run whatever we provide within the chroot filesystem, which is generally next to nothing.

```
david@fred: /home/david
16884
19457:66.342400 IP 64.12.24.36.5190 > 66.107.190.118.2441: ack 29 win 16384
19457:66.354913 IP 64.12.24.36.5190 > 66.107.190.118.2441: P 66:262(262) ack 29
win 16384
19457:67.013124 IP 66.251.128.114 > 66.107.190.118: IP 66.251.105.180.2861 > 66.
251.158.187.135: S 47346850:47346850(0) win 84240 (msg 1480, rtp, rtp, ackOK)
19457:67.013317 IP 66.107.190.118 > 66.251.128.114: icmp 60: 66.107.190.118 prot
ocol 47 unreachable
19457:67.107784 IP 66.107.190.118.2441 > 64.12.24.36.5190: ack 262 win 16688
19457:67.227557 IP 66.107.190.118.4569 > 66.251.133.7.4958: UDP, length: 23
19457:67.227561 IP 66.107.190.118.4569 > 66.251.133.7.4958: UDP, length: 12
19457:67.227597 IP 66.107.190.118.4569 > 66.251.133.7.4958: UDP, length: 12
19457:67.227612 IP 66.107.190.118.4569 > 66.251.133.7.4958: UDP, length: 12
19457:67.536591 IP 66.107.190.118.4520 > 66.250.66.190.4520: UDP, length: 8
19457:67.536595 IP 66.250.66.190 > 66.107.190.118: icmp 44: 66.250.66.190 udp po
rt 4520 unreachable
19457:67.618989 IP 66.251.128.114 > 66.107.190.118: IP 66.50.6.190.3051 > 66.251
.158.215.445: S 3189593942:3189593942(0) win 8760 (msg 1480, rtp, rtp, ackOK)
19457:67.619294 IP 66.107.190.118 > 66.251.128.114: icmp 60: 66.107.190.118 prot
ocol 47 unreachable
19457:68.536782 IP 66.107.190.118.4520 > 66.250.66.190.4520: UDP, length: 8
19457:68.576588 IP 66.250.66.190 > 66.107.190.118: icmp 44: 66.250.66.190 udp po
rt 4520 unreachable
```

Dumping packets from a network interface is not particularly useful unless you're using promiscuous mode.

```
david@fred: /home/david
Pred: iptables -nL
Chain INPUT (policy ACCEPT 334K packets, 72M bytes)
pkts bytes target prot opt in out source destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target prot opt in out source destination

Chain OUTPUT (policy ACCEPT 334K packets, 50M bytes)
pkts bytes target prot opt in out source destination

Pred: ■
```

The 'filter' table within iptables is split into a number of separate chains, which enables comprehensive configuration.

```
david@fred: /home/david
IPTABLES(0)
NAME iptables - administration tool for IPv4 packet filtering and NAT
SYNOPSIS
iptables [-t table] [-m] chain rule-specification [options]
iptables [-t table] -I chain [rulenum] rule-specification [options]
iptables [-t table] -R chain rulenum rule-specification [options]
iptables [-t table] -D chain rulenum [options]
iptables [-t table] -I [0-2] [chain] [options]
iptables [-t table] -N chain
iptables [-t table] -X chain
iptables [-t table] -P chain target [options]
iptables [-t table] -t old-chain-name new-chain-name
DESCRIPTION
Iptables is used to set up, maintain, and inspect the tables of IP
packet filter rules in the Linux kernel. Several different tables may
be defined. Each table contains a number of built-in chains and may
also contain user-defined chains.
Each chain is a list of rules which can match a set of packets. Each
Manual page iptables(8) line 1
```

iptables has a wide range of options, and all but the latest matches and targets are covered in the manual pages.

As default, the firewall is empty and will allow packets to flow in, out and through the firewall. Within the kernel, there are three filtering processes, each of which impacts packets at a specific part of the routing process.

For packets that are destined for an IP address bound to the local system – for example network services such as SSH or *Apache* – the INPUT chain is used to match and filter packets. Packets that originated on the local system are filtered in OUTPUT, and anything that is being routed in one interface and out through another is managed through the FORWARD chain. There's also a 'nat' table containing additional rule lists, where packets are modified rather than filtered, most commonly used to provide IP Masquerading capabilities where multiple hosts are located behind a single public IP address.

A typical firewall configuration that limits inbound packets and provides NAT capabilities would look something similar to the following:

```
# iptables -A INPUT -i lo -j ACCEPT
# iptables -A INPUT -i eth1 -j ACCEPT
# iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
# iptables -A INPUT -j DROP
# iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -o eth0 -j MASQUERADE
```

Each of the above commands adds a rule to iptables, which immediately becomes active in the kernel. We permit traffic from the loopback interface and eth1, our internal LAN, on to the firewall for maintenance, as well as allowing any outbound connections from the network to return into the firewall. This also takes care of nasty protocols such as ftp, which have multiple connections going in either direction, as we can track them using the 'RELATED' state match. Anything else entering the firewall is dropped to the ground, although it's often useful to log these packets using a 'LOG' target for analysis.

We also allow any system on our internal network, 192.168.1.0, to access the Internet through the firewall and use the firewall's public IP address. This is useful for situations where an ISP only permits one host to be connected to the Internet, or where IP addresses are in short supply.

Even with a firewall, there are often cases where potentially vulnerable

PROMISCUOUS MODE ETHERNET

See everything that's happening on your network

The Linux kernel is reasonably smart when it comes to handling IP packets and Ethernet frames, having kernel-level configurations to prevent junk or misdirected packets from entering the system. Within the IP layer, this is known as 'Return Path filtering', or simply 'anti-spoofing', where the kernel will drop packets coming in an interface that, according to the routing table, the source host does not live on. This can be turned on and off through `/proc/sys/net/ipv4/conf/<interface>/rp_filter`. Unless the network is handling complex asymmetric routes, leave it turned on.

Within the Ethernet layer, the kernel will drop everything not destined for our MAC address, other than broadcast traffic, so if we are connected to other hosts via a hub, the interface will never see packets for these other hosts.

While this makes perfect sense, for traffic sniffers and IDS devices, monitoring packets for other hosts is a basic function, so the kernel needs to know we want to see everything on the internet. We can tell the interface we want to see everything on the network by enabling promiscuous mode, and everything we monitor on the interface

will be actually what touches our wire. Turning on promiscuous mode is simply done using `ifconfig ethX promisc`, although many tools, such as `tcpdump`, will turn it on and off automatically for us when needed.

Using promiscuous mode is vital when you're using a span or a tap from a network device or uplink, because the interface won't have an IP address, and so it will never actually send out any traffic. With promiscuous mode turned off in this environment, other than broadcast traffic, we're not going to see anything exciting.

network services have to be accessible to the outside world. There are always network services that have a poor reputation for security, and until the last few years the popular and widely deployed DNS server BIND was one of the worst processes to have running on a host. Unfortunately, if we're running a DNS server, we can't simply running a DNS server, we can't simply firewall it, so managing the risk and ensuring we're up to date with package changes and vulnerabilities is an important start.

Managing risk

Tools such as *nmap* and *nessus* give us a good overview of what we're running and what may cause problems, particularly as the latter contains a database of known vulnerabilities and exploits for services. Running each of these against a network quickly alerts administrators to packages that may expose the internal network to security risks, as well as providing a list of everything we're actually running that is network accessible.

Firewall configuration flaws aren't completely unheard of, and opening up port 8080 rather than 80, if 8080 is running a proxy server, quickly causes problems on the network and allows a security hole to be opened.

You can also make use of the popular package *Snort* to manage traffic on the network by analysing the data structure of packets to look for exploits and attacks against vulnerable services. *Snort* can be run on the same device to monitor traffic inbound from the Internet. There is always a very fine line between what is open and what is blocked on a firewall, and *Snort* can be used to plug many of the holes in the firewall model.

As standard, *Snort* simply reports on packets and connections associated with attacks, but modified versions of *Snort*, such as *snort-inline*, will automatically add firewall rules to block hosts that attack the network from being able to hit any more ports. While this may seem like a good idea at first, it does take some tuning to avoid someone who is simply viewing a website, or trying to remember which port the mail server runs on, from being filtered on the firewall and having to be manually removed.

For wireless LANs, the *airsnort* application provides similar capabilities, but it includes support for guessing the key used by encrypted packets by making use of the flaws in WEP, which can often prove very useful.

Managing network security using Linux is no easy feat, and there are numerous distributions and packages available, including *Smoothwall* and *IPCop*, which provide a foolproof firewall installation for those who are not comfortable managing the kernel tools themselves. However, building a firewall and really understanding how the routing and packet filtering is functioning can provide great insight into where security holes lie within the infrastructure, as well as making it easier to track down any attacker.

Using NAT enables an internal network to be hidden behind our firewall, avoiding the need for large, public IP allocations.

The *nmap* tool, which is bundled with many distributions, is a great method for reviewing the open ports on a system.



REMOTE DESKTOPS

With remote X Terminals and desktop applications, you can use your local machine from anywhere. Here's how you can build an infinite desktop...

Connecting to a remote system through the shell is one of the many aspects of Linux that makes it so powerful. Anything that can be achieved locally, from reading email to posting news articles, can also be done from a remote shell connection using the bare minimum of bandwidth. However, despite all this power, it's sometimes just easier to use a GUI. While the X Windows protocol is obviously tailored for use over a network, it's particularly bandwidth hungry and inefficient, and also requires a version of X to be installed at both ends of the connection. There are several alternatives, all designed to bring remote applications to a local machine, either by encapsulating the X protocol or by implementing their own.

The most widely known remote desktop protocol is Virtual Network Computing, better known as VNC. The origins of the protocol can be traced back to a piece of hardware that operated as a very thin client, called a Videotile. VNC was originally intended to be a software version of this hardware, which is where the Virtual comes from in its name. VNC is a very simple protocol based around a client-server model and, in basic terms, it does nothing more than transfer a rectangle of pixels from the server's framebuffer to the client's display,

simply transmitting the changes from one moment to another. This would take up a large amount of bandwidth, almost akin to streaming a movie, so the transmitted data is compressed using a variety of methods.

Remote persistence

One of the most useful aspects of VNC is that no state is stored at the viewer. The viewer really is acting solely as the display – if it crashes or you quit, the session is left exactly as you left it running on the X server of the remote machine. This means that when you next connect to the server, the display is the same as if you'd simply turned the screen off, even down to the position of the cursor. This works well when you need a desktop to be available from a variety of locations, either on various machines on the local LAN, or from the other side of the world across the Internet.

Another advantage to the VNC protocol is that because it's based around simply transmitting what are basically screenshots, it isn't reliant on any particular graphical environment. That means you could just as easily have a remote Linux desktop on a Windows machine as you could a Windows desktop on a Linux machine. The only difference is that thanks to X Windows, you can launch as many totally separate X session for VNC, whereas with Microsoft Windows you need to share the current root desktop. The VNC session can also be shared with other users, either as an interactive session, or as view-only.

As the protocol is so simple, several clients have made their way onto a variety of platforms, from Windows CE to embedded Java running inside a web browser. VNC has also become the default remote desktop protocol for KDE and, more recently, for the latest 2.8 release of Gnome.

While the original VNC team can still be found developing RealVNC, providing both a commercial version and a free version, being an Open Source project has meant that there has been several forks in the source code that aim to address some of

VNC's perceived problems. Of these, the most popular is called TightVNC, a version of VNC that successfully manages to compress the protocol more effectively, and can even compress using JPEG, making it more usable over restricted connections.

Another interesting VNC fork is called gemvnc. Instead of starting a new X Windows session, it shares the current framebuffer in the same way that the VNC server for Microsoft Windows does. This makes it easier to manage a single session, and also allows for better remote control.

Security

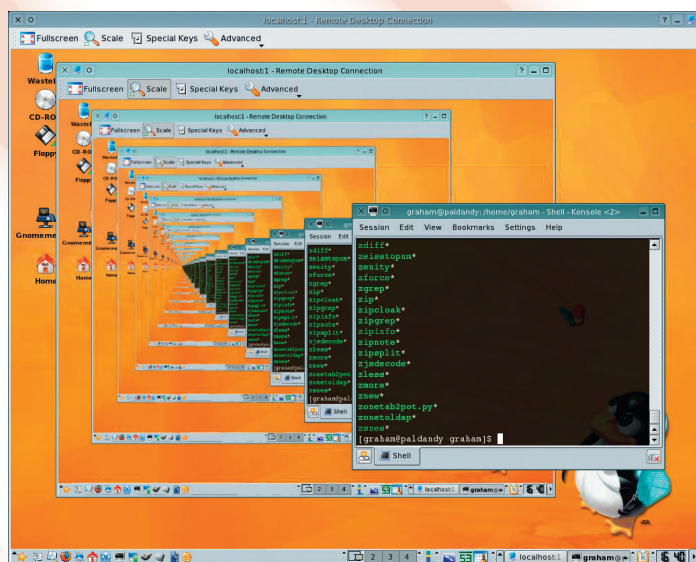
There are three main security issues with a standard VNC installation. Firstly, while the passwords aren't transmitted as Plain Text, it's still possible to intercept the encrypted version and attempt a brute force attack on the server. Strong passwords are the best defence for this attack, with at least eight characters recommended.

However, perhaps the biggest problem with VNC is that, by default, all network traffic between the client and the server is also unencrypted, meaning it has the potential to be stored and reassembled by a third party. This can be solved using the ever-reliable SSH, along with port-forwarding. To funnel the connection from the remote server to the local client through SSH, use:

```
$ ssh -C -f -L 5901:localhost:5901
remotehost vncviewer :1
```

This forks an SSH session into the background (asking for a password, unless the appropriate keys have been shared), and tunnels the VNC session from the external 5901 VNC port, through SSH (on port 22) and locally re-attached to port 5901, which is why vncviewer :1 works by connecting to the localhost.

A modern development that competes directly with VNC is called NX, developed by NoMachine and recently brought closer to the world of Open Source with the generous GPL release of its server component. NX attempts to make up for the shortfalls of VNC, and it does both very well.



Recursive desktops using VNC: you are feeling veeeeery sleepy...

The best thing about NX is that it builds on the power of X Windows by using the same X protocol, but addressing X's shortfalls through a mixture of differential compression and a proxy X server. The compression is handled by the NX proxy, which is responsible for servicing the client and server requests, and reducing duplicate and redundant data before it needs to be transmitted across a network. Running as a client, the proxy encodes X client requests and decodes the corresponding replies; as a server it does the exact opposite. NX can also translate and tunnel other protocols, such as RDP and VNC.

Screen updates are usually translated into X protocol requests, but differential compression means that it doesn't arbitrarily compress whole sections of the display, as with VNC, but can instead stack similar requests (by comparing MD5 checksums), drastically reducing their number. NoMachine claims that after years of tuning, its proxy cache can handle 60% to 80% of requests, and with common office tasks such as servicing fonts and images, cache hits can be as high as 100%.

Other improvements to protocol efficiency are made with image compression and streaming. Image compression is configured by the NX agent, and can use JPEG, PNG, RDP, TIGHT or ZLIB, with the clever part being the intelligent caching of the image. The local and remote caches only update the image data when there's no other X protocol messages that need encoding, in an attempt to keep the system as responsive as possible. Images of a certain size are stored in a persistent cache so that graphical information that's common to each session, such as backdrops, can load off the local cache. NX has a limited amount of control over the bandwidth that can help to avert long delays in user interaction by manually reducing the TCP window when bandwidth has

become saturated, as often happens with a screensaver, for example.

Despite there being no official connection with KDE, the KDE team have taken this new technology to its heart. Not only are there ambitious plans to integrate NX into the heart of KDE, potentially offering an NX KPart, 'nx://' kio-slave and KWallet storage, but there are also plans to use an NX server at the core of the development cycle. Rather than each developer having to laboriously rebuild the current CVS version every night, a single server could be kept continually up-to-date, providing desktop access to over 200 sessions.

Generation NX

With the latest major release (1.4) of NX, NoMachine introduced the much requested 'persistent sessions' feature, which enables a user to suspend a session when closing the NX client and resume from the same point when the client reconnects. This is similar to VNC, but while a VNC session continues as an independent X session on the server, the NX session is hibernated, meaning that any programs running on NX won't be able to execute time-dependant code while the session is hibernated. Security is based around SSH tunnelling and encryption, using private and shared keys to authenticate the client with the server, making NX as secure as VNC when tunnelled through SSH.

Another important aspect to a remote desktop is file sharing and printing, which is conveniently handled by using shared folders and printers in *Samba*. While it would be easy enough to configure this manually, having it easily set-up and piped through a secure channel is a great advantage.

While all this sounds great on paper, it's performance in the field

LINUX TERMINAL SERVER PROJECT

The king of Linux thin clients

The Linux Terminal Server Project (LTSP) is an Open Source solution for running thin terminals connected to a Linux server. The diskless thin clients can be machines that were getting old six years ago, such as a Pentium 133 with 32MB RAM, PCI video, a decent monitor and a network card. Obviously, on its own, this wouldn't be capable of doing much, but with the help of LTSP and a properly configured server, the clients can run resource hogs. The actual running is done on the server. Depending on system load, the clients should be as responsive as if you were running the apps while sat at the server's screen.

Other advantages to LTSP include lower running costs because there are far fewer moving parts in diskless clients, and maintenance is much easier because all software is installed and configured from the central server.

One of the most successful areas for LTSP deployment has been in schools, where plenty of low-cost or donated workstations can be run, all connected to a single server. Not only does this have the advantage of introducing students to open standards, but it also means that they can run the same software at home, without any of the licence or cost issues.



An NX client running inside a VNC session.

that counts, and that's where NX shines. Using KDE over an ordinary 512Kbps connection, the interface is almost as responsive as sitting at the actual machine. Hopefully it won't be too long until there's an open NX client and server, making this powerful remote system more accessible to everyone.



DISTRIBUTED COMPILATION WITH DISTCC

Speed up project development by turbo-charging your build phase. We show you how...

Compiling a typical C or C++ program is a task that generally exhibits some parallelism: any non-trivial program will consist of multiple source code files, most of which could be independently compiled to object code. As such, this is a job ripe for exploiting the power of distributed, parallel computing.

There's a well-proven tool for distributed compilation: *distcc* (see <http://distcc.samba.org>). It's simple, effective and already shipped with many popular distros. *distcc* is employed by many high-profile projects, including *Samba*, to speed up compilation, and was even used to produce id software's *Doom3*.

How does it work?

The *distcc* package consists of a daemon, *distcc*, and a compiler-front-end, called *distcc*. You install and run the daemon on each machine that you wish to take part in your compile farm. It can be launched from a system's start-up scripts, or you can use the Internet superserver to run it on demand. Then all you have to do to distribute compile jobs to the nodes running *distcc* is call your compiler with the *distcc* wrapper. For instance:

```
distcc gcc some_code.c -o some_code.o
```

The magic is performed by *distcc*. What happens is that it recognises that this is a command to compile a source

file to object code (only compilation can be performed remotely; linking is done locally). So, first it calls *gcc* locally to pre-process the source file, which is 'some_code.c' here. Then it looks for a free 'slot' in the machines available to it to do the compilation.

If a slot is available locally, the pre-processed file is compiled to object code normally. If the slot is available remotely, the pre-processed code is sent to that machine, *gcc* invoked remotely (via *distccd*) to create the object code, and then the object code passed back, where *distcc* writes it to disk. The end result is the same, wherever the actual compilation is done. Here we get an object file called 'some_code.o'.

Actually, there's more to it than this. First note that each node in the compile farm must have the same version of *gcc* available under the same executable name. Typically, */usr/bin/gcc* will be a symlink to your default compiler, but this default may be different on different machines. The solution is to be more explicit when naming the compiler to use. Modern versions of *GCC* will be available with a binary name, which unequivocally identifies the target and version, such as 'i386-linux-gcc-3.4'. If this doesn't exist, you can easily create a symlink.

The other critical issue with *distcc* is with makefiles. To do parallel compilation with *distcc*, you employ GNU make's ability to execute targets in parallel (with the *-j* switch); *distcc* knows nothing about parallelism itself. Therefore, you must be careful about specifying dependencies when creating your makefiles. Makefiles generated by automake will typically parallelise with no problems.

Using distcc

Once the *distcc* nodes are ready, the next step is tell *distcc* on the compile host what nodes to use. This is done with the *DISTCC_HOSTS* variable. For example, try this code:

```
export DISTCC_HOSTS="localhost
tyr odin/1"
```

You should see the *distcc* man page for details but typically, a whitespace-separated list of hosts is enough. By default, *distcc* will use two slots (execute two compilation jobs) on each host. In the above example, 'odin/1' tells it to use only one slot on the host odin (this machine is short on memory, so I don't want it compile two jobs at once). Note that if you want local compilation to take place, you must include localhost in the list.

Also remember that the localhost has extra overheads, so if you have many machines in your cluster, omitting localhost may actually be beneficial. A rule of thumb is that if the local machine has less than a fifth of the total computing power of the cluster, leave it out.

If your project creates makefiles using a configure script, you should next call *configure*, explicitly stating the compiler to use, prefaced by 'distcc'. Pass all other config options as normal. For example:

```
CC="distcc powerpc-linux-gcc-3.3" ./configure --prefix=/usr --enable-coolfeature
```

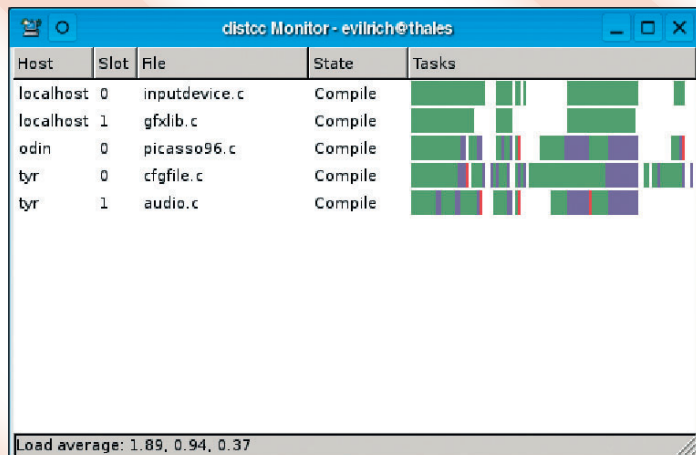
You should then make with:

```
make -j5
```

If you don't use the *-j* switch, the build will proceed sequentially as normal. The number specified by this switch is the maximum number of jobs to execute in parallel. Here we state 5: we have the default two slots, each on localhost and tyr, and one slot on odin.

Now for some numbers. Compiling *E-UAE*, a version of the Unix Amiga Emulator that I've been developing, just on my main Linux/PPC development machine (a lumbering PowerMac 9500 with a 333MHz G3), takes a shade over 15 minutes. Adding tyr, a 600MHz G3-powered Pegasos, and odin, a Starmax 3000 with a 233MHz G3, brings the compile time down to just over eight minutes, which is clearly a significant improvement. And this is with build scripts that poorly exploit parallelism! With better-designed makefiles, I'm sure that this could be reduced even further.

The graphical *distcc* monitor displays the status of each of the machines in our compile farm.



Host	Slot	File	State	Tasks
localhost	0	inputdevice.c	Compile	
localhost	1	gfxlib.c	Compile	
odin	0	picasso96.c	Compile	
tyr	0	cfgfile.c	Compile	
tyr	1	audio.c	Compile	

Load average: 1.89, 0.94, 0.37

PARALLEL COMPUTING WITH PVM

Discover the best way to build a render farm with PVM and *POV-Ray*.

PVM (Parallel Virtual Machine) is a software framework that enables a network of heterogeneous computers to be bound together as a single, distributed parallel processor. It consists of two main components: a daemon, *pvmd*, which runs on each of the nodes in the network and controls jobs on that node; and *libpvm*, a software library that can be used to create software and which can make use of such a cluster of machines.

PVM can be effectively employed for many kinds of number-crunching tasks, but we're going to illustrate its use here to do distributed raytracing with *POV-Ray*, the popular, freeware raytracer (see www.povray.org).

There are a number of ways to distribute rendering with *POV-Ray*, and using PVM for this is experimental and doesn't technically achieve the best results. However, PVM is simple to use and is interesting because it's portable and can be employed for many other types of problem. Note that *POV-Ray* doesn't include PVM support by default. You must build it with the PVMPOV patch (available from <http://pvm pov.sourceforge.net>).

PVMPOV works by splitting up the render into chunks and parcelling the jobs out to the known PVM hosts. It executes *POV-Ray* as slaves on the remote hosts, via the *pvmd* daemon.

Building the cluster

We aren't going to dwell on installation issues or cover how to use *POV-Ray* itself. You do need the PVM daemon installed on or available to each node of your parallel cluster, and the PVMPOV-patched version of *POV-Ray*. We used Debian while testing, which has precompiled packages of both.

The PVM cluster can be managed by the management console, *pvm*. Alternatively, you can use the TCL/Tk-based *xpvm* GUI. This enables you to add and inspect hosts, manage jobs, and so on. You can specify the PVM nodes to use by passing a Plain Text file to PVM. For example:

pvm hosts

The 'hosts' file can contain just a list of the hosts to use, separated by

newlines. You can also specify various options per host. See the *pvmd* man page for more details.

When you start PVM, it will launch *pvmd* by remote-login on the hosts it knows about. You can also add hosts from the PVM console with the *add* command. To be able to start *pvmd* remotely, you need an account on the remote machine. Typically, it'll call *rsh*, but on modern set-ups this will most likely be *SSH*. Anyway, you should avoid having to use passwords, so edit your */etc/hosts.allow* file or, if using *SSH*, use key-based authentication.

You can test that *pvmd* is working on your nodes by issuing the 'conf' command from PVM. You'll see something like this:

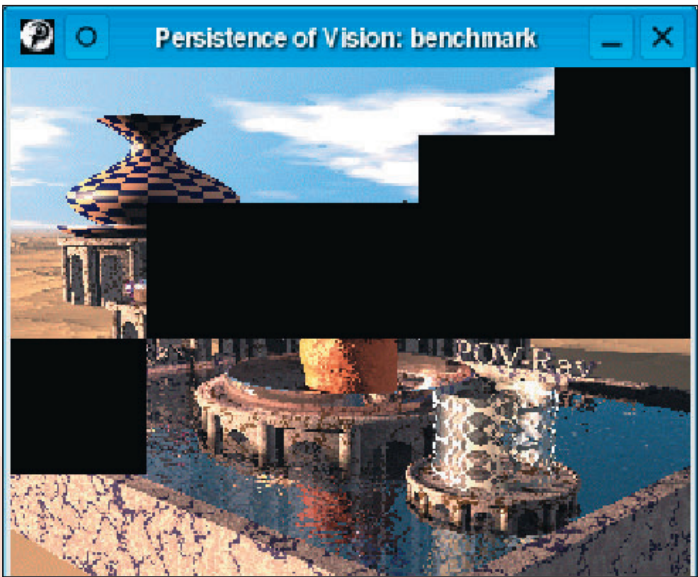
```
pvm> conf
3 hosts, 2 data formats
      HOST  DTID  ARCH
SPEED  DSIG
confucius.home.localnet  40000
LINUX  1000 0x00408841
      odin  80000
LINUXPPC 1000 0x0658eb59
      tyr   c0000
LINUXPPC 1000 0x0658eb59
```

To test the running of PVM jobs, try using the 'spawn' command. For example, to run *POV-Ray* on *odin*, you should execute the following from the *POV-Ray* console:

```
spawn -odin -> povray
```

If successful, you'll see the console output from *POV-Ray*. If it says 'No such file', it can't find the executable to run. *pvmd* looks for executables relative to the path *\$PVM_ROOT/bin/<ARCH>* on each host and you have to install any executables you want to run via *pvmd* in that path (or specify the path in another way).

\$PVM_ROOT is set by default by your installation – on Debian it's */usr/lib/pvm3*. The *<ARCH>* part of the path identifies the architecture of the PVM node. For example, 'LINUX' for a x86 Linux box or 'LINUXPPC' for a PowerPC box (this is so you can have executables for various platforms installed under the same tree shared by NFS). On a current Debian/PPC setup, the install is broken and you'll have to rename */usr/lib/pvm3/bin/LINUX* to */usr/lib/pvm3/bin/LINUXPPC*.



You can see by the way that the render is created that the *POV-Ray* scene is divided up into blocks.

Render time

If you have your PVM cluster ready and the patched *povray* is installed then it's almost time to start rendering. The final step is to make the *POV-Ray* data files available to each machine in the node. Typically, this can be done by creating a working directory containing the POV scene description and any include files, and mounting that under the same path on each of the PVM hosts via NFS.

On the master machine, we can then *cd* to the working directory and issue (from a plain shell, not from the PVM console):

```
povray +lbenchmark.pov +Ooutput.
tga +N +NT6 +NW64 +NH64 +v
```

The *+N* option enables PVM support, while the *+NT* option specifies the number of PVM jobs to use. Here we use six, two on each host. PVMPOV splits the render into blocks. The *+NW* and *+NH* specify the height and width of each block. Other standard *POV-Ray* options are employed as normal.

The quality of the render isn't as high with PVMPOV as with other methods, and load-balancing is primitive. Nevertheless, experimenting with PVM is a lot of fun, and can drastically speed up test renders of scenes with *POV-Ray*. **LXF**

NETWORK GAMING

The most popular distributed application on your average home network is network gaming, and there are dozens to choose from, whether it's FPSes such as *Doom* and *Quake*, strategy games such as *FreeCiv* or traditional games such as chess.

How you run such a game on your network depends highly on the game in question. We'll examine *PrBoom*, a modern version of the classic shooter. Linux and Windows versions of this are available, but since it uses *SDL*, it should work on just about any modern OS supported by *SDL*.

Like many other network games, *PrBoom* is separated into server and client parts. The server is a console application, which can quite happily run on any box and takes the role of co-ordination. For a networked game, the server is started first and the client's machines log in to the server to play. The server can be run on the same machine as one of the clients.

The *PrBoom* server can be started with the command *prboom-game-server*. Options specify what ports to listen on for client connections and game level options. Clients must specify the host to connect to:

```
prboom -net odin
```

If the data file (WAD file) for the level specified by the server isn't available to a client, it will download the necessary files from the URL specified by the server via *wget*. This means that you don't need to install the same WAD files on each of the client machines.