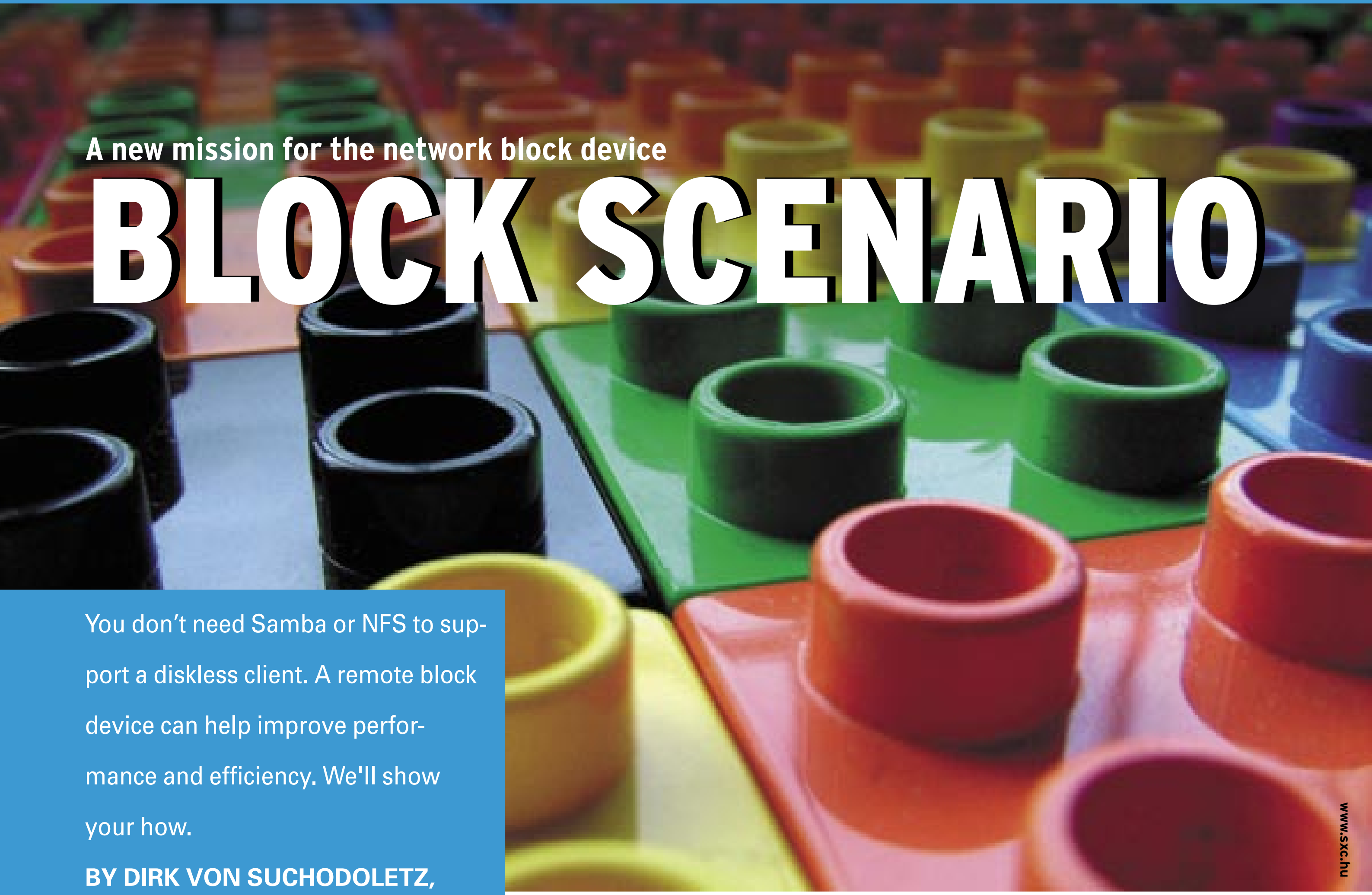


A new mission for the network block device

BLOCK SCENARIO

You don't need Samba or NFS to support a diskless client. A remote block device can help improve performance and efficiency. We'll show your how.

BY DIRK VON SUCHODOLETZ, THORSTEN ZITTERELL

Experts are well aware that the traditional PC networking model wastes valuable disk space with redundant data storage. Desktop computers typically have one or more local hard disks, although the operating system and the applications that run on the desktop may be identical throughout the organization.

An alternative to this traditional approach is to use centralized data management with diskless clients. The diskless client option assumes fast and reliable network connections and powerful servers.

A typical diskless client downloads any data through a network filesystem. This scenario places unnecessary load on servers and networks. When a server on the network distributes a filesystem that the clients do not modify, most of the functionality associated with the file system is unused. One interesting alternative for better performance with diskless clients is to provide a root filesystem using a network block device rather than a network-based filesystem such as NFS.

The starting point in our quest for a high-performance root filesystem for the LAN was a pool of 60 Linux-based diskless clients (Figure 1), which were deployed at the University of Freiburg three years ago. The previous environment, which was used in three tuition pools, relied on two NFS servers, with setup and configuration tasks handled at client runlevel start.

Logging In

By the time a user had negotiated the KDE login, each client had downloaded

Figure 1: The machines in the CIP pool are set up as diskless clients. To avoid overloading the network at boot time, we dug deep into our box of tricks and replaced the network filesystem with remote block devices.

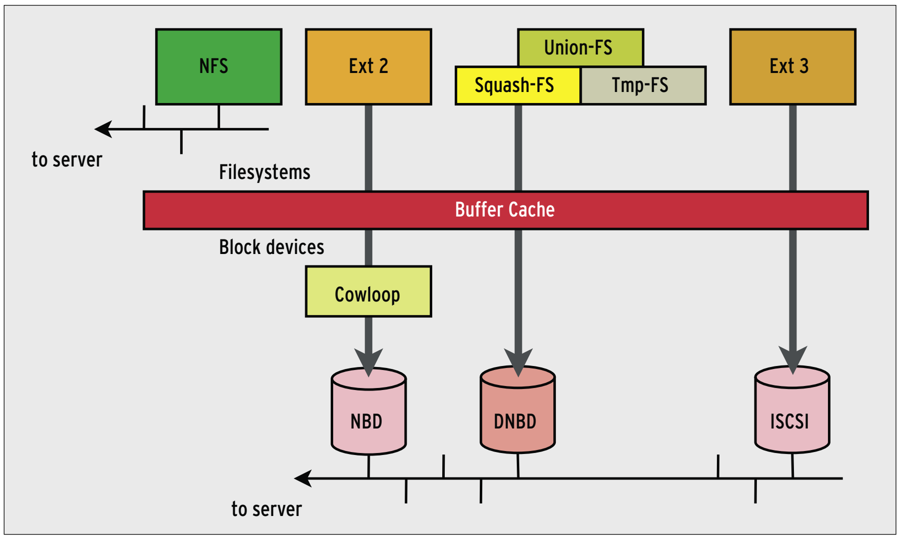


Figure 2: Server and client can exchange files and or blocks on multiple levels. The figure shows the data interpretation, but not the software implementation. The special LUFS (Linux Userland Filesystem) implementation, for example, can mount an FTP server as a filesystem.

up to 350 MBytes from the server, which would take at least a minute. As you can imagine, booting 15 machines simultaneously was fairly taxing on server resources. To address this problem, we started a search for potential optimizations, but we optimized without touching the server and network hardware.

We soon discovered that the client boot behavior had a major affect on performance. Wherever it made sense, we decided to handle configuration in the initial ramdisk and parallelize the steps. We also investigated alternative network filesystems and network block devices with a suitable filesystem. A combination of these two approaches reduced the volume of data we needed to transfer over the network from 350MBytes to less than 50, and this, in turn, slashed boot time by almost half.

Legacy NFS

The Linux kernel doesn't really mind where the root filesystem is located: on a local hard disk, a SCSI RAID drive, a flash device, or a remote network server. In the latter case, Linux typically resorts to the NFS network filesystem. NFS is widespread and simple to configure, and it has been part of the kernel for years. But NFS is starting to show its age: prior to version 3, there was no security. The server blindly trusts client-based user authentication and distinguishes clients by their IP addresses only.

NFS is RPC-based (Remote Procedure Call) and uses remote function calls to handle any kind of access. From the developer's point of view, this is an elegant approach, but one that suffers from serious overhead, especially if you are handling small files. Users with an NFS home directory might be able to live with this, but if you have a root filesystem shared by a larger number of clients, the numbers really start to tell. The System V boot procedure, with its myriad of tiny start scripts, is particularly prone to overhead.

While we were searching for a faster approach, it became apparent that our requirements – scalability, performance, data integrity, and availability – were contradictory. For example, caching tends to improve scalability and performance while affecting data integrity. Without additional measures, you cannot guarantee that the data in the client's cache is identical to the data on the server. And although synchronization will improve integrity, this again affects scalability.

SAN and Block Devices

This left us with an escape route via the lower levels: instead of centralizing the filesystem, you can remotely control hard disks at block device level (see the box titled "Blocks and Files"). This principle is popular with Storage Area Networks (SAN), which are typically based on LAN-independent fibre channel fabric. This said, the technology is expen-

Blocks and Files

Filesystems control access to files; block devices typically abstract the hardware (hard disks, for example) and support block data exchanges (see Figure 2). The two meet when a normal filesystem is based on a block device to support file storage in blocks. A modern operating system will enhance performance by caching block transfers in RAM.

The available network filesystems differ in the file types and sizes they support, and with respect to access control. For example, Samba does not support special Unix files such as sockets or named pipes in the default configuration, and this can trip up some applications and Linux GUI desktops. Block devices support the use of arbitrary block-based filesystems, no matter whether the block device is mounted locally or is remotely accessible over a network. This gives you a far more choice than you get with network filesystems.

No Write Access

Just like with a remote filesystem, a network block device can be exported to support read-only or read/write access. There is a major difference when you are looking at shared write access to a resource: network filesystems provide locking mechanisms or coordinate file access to prevent damage to files. Network devices put much less effort into this and are quite happy as long as read or write access is restricted to individual blocks. The only protection they provide is to ensure that block transfers are atomic.

Block devices are not particularly interested in whether the blocks actually belong to a filesystem – they are blissfully unaware of overlying data layers and the way they are organized, and this can make it difficult to support distributed use of a network block device with a normal block-based filesystem. If multiple clients were to attempt to write to a shared network block device, they would be unaware of their peers, and they would destroy each other's data structures due to a lack of coordination. Local caching on the clients makes this chaos perfect and leaves the filesystems with unintelligibly trashed data.

This is not a major concern with diskless operations. As long as nobody attempts to write to the network block device, any number of clients can share it. But for write access you need to prevent a second client from mounting a block device if another client wants to write to it.

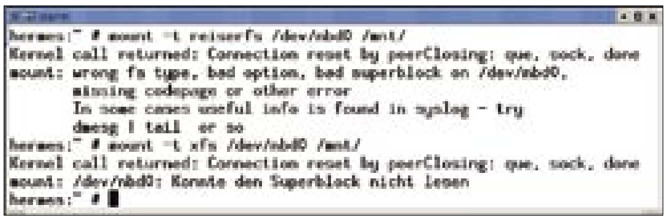
sive. There is the ISCSI industrial standard for less money, and the Linux kernel has driver support for it, but ISCSI overhead is just too much for a normal desktop.

A cheaper way to avoid the problems of NFS by operating at the block device level is through the Linux network block device (NBD) feature. The mainstream kernel has had Network Block Device (NBD) code for four years now. The module that manages network block devices is fairly simple, leaving integrity checking of the transferred data to the underlying TCP layer.

Using NBD

You can run *modprobe nbd* to tell you if your kernel has an NBD module configured. You should not see an error message. The accompanying userspace tools, such as */usr/bin/nbd-server*, are provided in a separate package by most distributions. Version 2.8.4 was current when this issue went to print; it is available from [2]. Suse Linux 10.0 still uses Version 2.7.4, which will do fine for our purposes.

For initial testing, you might like to eliminate the possibility of network



```
hermes:~ # mount -t reiserfs /dev/nbd0 /mnt/
Kernel call returned: Connection reset by peerClosing: que, sock, done
mount: wrong fs type, bad option, bad superblock on /dev/nbd0,
       missing codepage or other error
       In some cases useful info is found in syslog - try
       dmesg | tail  or so
hermes:~ # mount -t xfs /dev/nbd0 /mnt/
Kernel call returned: Connection reset by peerClosing: que, sock, done
mount: /dev/nbd0: Konnte den Superblock nicht lesen
hermes:~ #
```

Figure 3: Our attempt to mount Reiser-FS on a read-only block device failed. The filesystem needs to add entries to the journal. Unfortunately, shared network block devices can't be writable.

problems and run the client and the server on the same machine. The easiest way to do this is with a free partition. If you don't have one, a container file with any filesystem will do the trick (Listing 1). We tested Reiser-FS, Ext 2, and XFS successfully in our lab. Formatting tools will tell you that the file is not a block device (Line 8) but you can just ignore the helpful advice.

In our example, the admin creates a 100MByte file (Lines 1 and 2), adds an Ext-2 filesystem (Line 6), launches the NBD server on port 5000 (Line 14), and mounts the device on the same machine in Line 15. The choice of port number is arbitrary; this would let you launch multiple servers for various containers. The NBD client needs the kernel module and expects the IP address and port number of the server followed by the name of the local block device.

Remote

To run the client and the server on separate machines, simply replace the 127.0.0.1 address in the *nbd-client* command line with the server IP. The *nbd-server* can export LVM volumes or partitions instead of simple container files. To do this, just replace the filename with the device name: *nbd-server 5001 /dev/sda4*.

The client closes the connection to the server as follows: *nbd-client -d device*, but make sure you unmount the block device before closing the connection to the server:

```
umount /mnt
nbd-client -d /dev/nbd0
```

As explained in the box titled "Blocks and Files," allowing multiple machines to write to a shared network block device causes chaos. In many scenarios, there is no need for this chaos caused by many devices writing to a shared block device. Read access is fine for the root filesystem on a diskless client, a shared application directory, or a document repository.

Read-Only

The *-r* option launches the block device server in read-only mode, preventing clients from modifying the block device. Of course, this also disqualifies journaling filesystems.

We were unable to mount Ext 3, Reiser-FS, or XFS via a read-only block device in our lab. Depending on the filesystem, error messages of varying severity were issued; Figure 3 shows what Reiser-FS had to say about this. Specifying the *-o ro* mount option did not improve things; journaling filesystems always want to write their journals. The SquashFS [7] and Ext 2 filesystems were fine, however.

As an alternative, the NBD server supports a *-c* option for copy on write (Listing 2, Line 1). To support this option, the server creates a file for each client,

Listing 1: NBD Test

```
01 hermes:~ # mkdir /exports
02 hermes:~ # dd if=/dev/zero
   of=/exports/nbd-export bs=1024
   count=100000
03 100000+0 records in
04 100000+0 records out
05 102400000 bytes (102 MB)
   copied, 0,987222 seconds, 84
   MB/s
06 hermes:~ # mke2fs nbd-export
07 mke2fs 1.38 (30-Jun-2005)
08 /exports/nbd-export is not a
   special block device.
09 Continue anyway? (y,n) y
10 Filesystem-Label=
11 OS-Typ: Linux
12 Block size=1024 (log=0)
13 [...]
14 hermes:~ # nbd-server 5000 /
   exports/nbd-export
15 hermes:~ # nbd-client
   127.0.0.1 5000 /dev/nbd0
16 Negotiation: ..size = 100000KB
17 bs=1024, sz=100000
18 hermes:~ # mount /dev/nbd0 /
   mnt
19 hermes:~ # ls /mnt
20 .  ..  lost+found
21 hermes:~ # df
22 Filesystem          1K blocks
   Used  Available Used% Mounted
   as
23 [...]
24 /dev/nbd0                96828
   13      91815    1%        /mnt
25 hermes:/ # time dd if=/dev/
   zero of=/mnt/text count=8192
   bs=1024
26 8192+0
27 8192+0
28 8388608 bytes (8,4 MB) copied,
   0,038112 seconds, 220 MB/s
29
30 real     0m0.046s
31 user     0m0.008s
32 sys      0m0.036s
```

where it stores that client's changes. Line 9 shows an example. When the client disconnects, the changes are lost, although the file itself is kept. The developers themselves say that the *-c* option's performance is not very fast, so you might prefer a client-side solution such as Union-FS or the COW module (copy on write).

Unfortunately, NBD's security design is rudimentary at the most. NBD security is similar to NFS security but far less flexible. Admins can specify a list of machines to which the server will grant access to the device. NBD totally lacks advanced features that provide services such as authentication, integrity checking, or encryption.

Targeted Writing

Unfortunately, diskless clients need to create or modify files at various positions on the filesystem at runtime – */etc/resolv.conf* or the various temp directories, for example. One solution involves using translucent filesystems such as Union-FS to support changes via a superposed ramdisk, however, these changes are not persistent, and the system reverts to its original state on rebooting.

Cowloop (Copy On Write Loopback Device [1]) is another possible approach for read-only block devices. Cowloop makes a block device writable by storing changes separately in a sparse file (Figure 4), which again can reside in a ramdisk. In comparison to Union-FS which needs to copy a whole file to the writable layer to reflect even the tiniest of changes, Cowloop is far more frugal with its use of space and only stores changed blocks.

Listing 2: Copy on Write

```
01 hermes:~ # nbd-server 5000 /
   exports/nbd-export -c
02 hermes:~ # nbd-client
   127.0.0.1 5000 /dev/nbd0
03 hermes:~ # mount -t xfs /dev/
   nbd0 /mnt
04 hermes:~ # ls -al /exports
05 insgesamt 100008
06 drwxr-xr-x   2 root root
   60 2006-04-03 12:39 .
07 drwxr-xr-x  23 root root
   4096 2006-03-25 21:16 ..
08 -rw-r--r--   1 root root
   102400000 2006-04-03 12:33
   nbd-export
09 -rw-------   1 root root
   270336 2006-04-03 12:39
   nbd-export-127.0.0.1-7812.
   diff
```

Cowloop

After unpacking, compiling, and installing the source code from [1], you should have a kernel module and a tool. You can use Cowloop in combination with NBD as shown in the following:

```
modprobe cowloop
cowdev -a /dev/nbd0 /tmp/nbd.cow
mkdir /mnt/nbd-rw
mount /dev/cow/0 /mnt/nbd-rw
```

This example links the network block device *nbd0* with the writable file */tmp/nbd.cow* and mounts the new block device. Write operations on *nbd-rw* do not affect the network block device. If Cowloop complains about a missing */dev/cow/ctl*, or */dev/cow/0*, the following commands should help silence those complaints:

```
mkdir /dev/cow
mknod /dev/cow/ctl b 241 255
ln -s /dev/cowloop0 /dev/cow/0
```

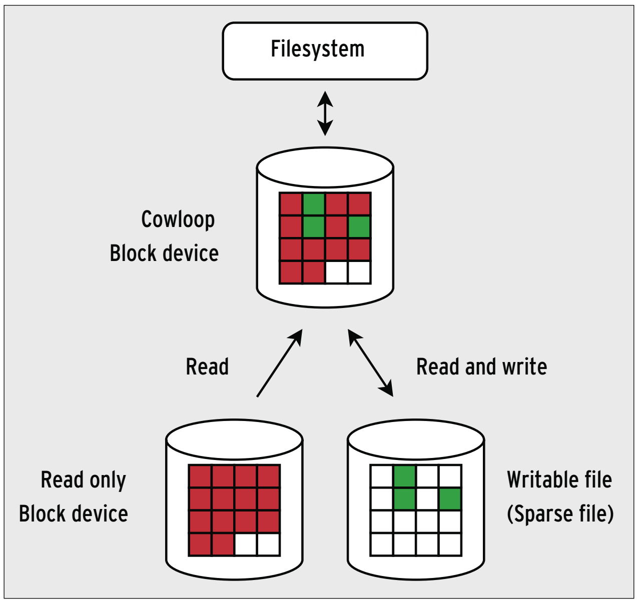


Figure 4: The Cowloop block device (top) draws on two resources: a write-only device (left) is the basis, and any changed blocks are stored in a separate file (right). The content of the original block device remains unchanged.

After unmounting the combined block device, *cowdev -d /dev/cow/0* will remove Cowloop.

In contrast to this approach, Union-FS runs at a higher filesystem level. Union-FS stores modified files in an additional filesystem, making modifications easy to trace. In fact, you can use standard Linux tools to search the superposed filesystem.

Memory Hook

Diskless clients have no local storage facilities and thus draw all their information from the network. Normally, the bootloader or a kernel parameter tells the kernel where the root filesystem is. But if the root filesystem is located somewhere on a LAN, you must provide some means for initially enabling the network

Table 1: NBD Variants

Name	Explanation
NBD	The Network Block Device is the predecessor of all Linux network block devices. At present, there is one active developer and a mailing list with a reasonable volume of traffic. [2]
ANBD	Another Network Block Device is a compatible NBD extension from 2003. It supports multithreading and promises better error messages than its predecessor. [5]
ENBD	The Enhanced Network Block Device is being pushed along by one active developer; there is a mailing list that sees a couple of messages a month. ENBD extends NBD adding an automatic restart if the connection is lost, authentication, and support for removable media. [3]
DNBD	The Distributed Network Block Device uses UDP as its transport protocol, and thus supports multicasting, client-side caching, and server redundancy. It only supports RO exports. [6]
GNBD	The Global Network Block Device is the basis for GFS (the Global Filesystem). [4]

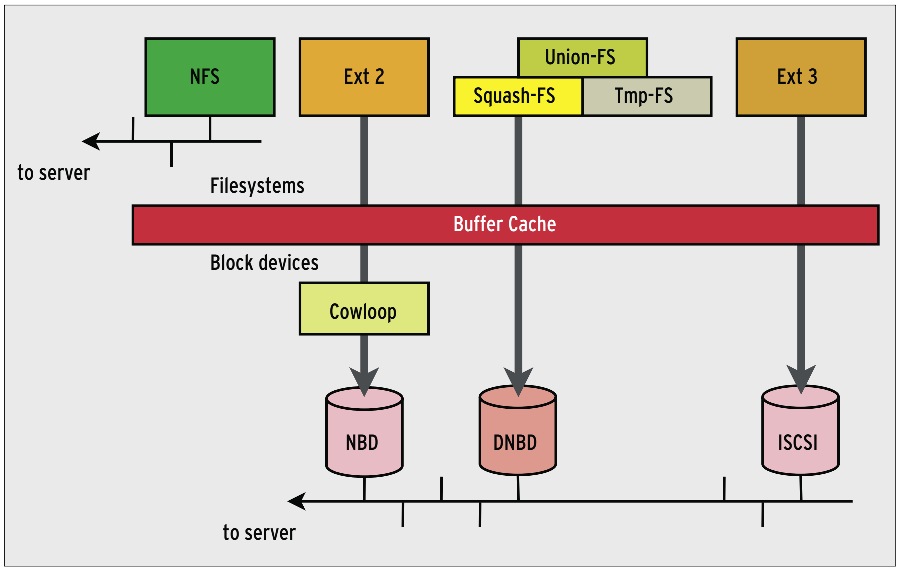


Figure 5: A combination of the techniques discussed in this article is possible. Block devices mostly have block-based filesystems; the exceptions are Cowloop or Union-FS, which provide a new writable layer based on block devices or filesystems.

interfaces and the IP configuration, both of which normally require a working root filesystem.

Three possible approaches to the problem of providing a root filesystem for enabling the initial IP configuration are as follows:

- **Kernel root over NFS** – Very early versions of Linux were capable of supporting "kernel root over NFS," which involves mounting the root filesystem directly via NFS. However, the kernel root over NSF option assumes that all the required components are built into the kernel, including the settings for the network card and for NFS, and let's not forget IP auto-configuration, which assumes a working DHCP client. Problems occur when you change the NIC or need to patch the driver. In both cases, you would need to rebuild the whole kernel.
- **Initial ramdisk** – An initial ramdisk is technically a block device in RAM. The ramdisk has a formatted filesystem with all the tools, scripts, and modules required for the setup to the point where the root filesystem is mounted. The initial ramdisk approach has its drawbacks, such as the effort involved in creating the initial ramdisk, overhead due to the block device and filesystem, and the complex transition from the small root filesystem on the initial ramdisk to the newly mounted target root filesystem. pivot_root handles the transition, and freeramdisk releases the ramdisk memory after the switch has occurred.
- **Early userspace** – A recent development, dubbed early userspace looks likely to replace both kernel root over NFS and the initial ramdisk approach in the not-too-distant future. InitramFS is a set of special kernel structures that map to a standardized root filesystem that can either be built into the kernel as Temp-FS (in a similar approach to the initial ramdisk) or kept and loaded separately as a CPIO archive.

Linux kernel 2.6.15 or newer no longer uses *pivot_root* to switch to the target root filesystem. Instead of using *pivot_root*, you can simply redirect the mountpoint to /. A tool titled *run-init* (located in klibc) deletes the data formerly stored at this point.

Alternatives

In addition to NBD, a number of other free implementations of network block devices are also available, such as ENBD [3], GNBD [4], or ANBD [5] (see Table 1). These block device drivers are not part of the kernel as of this writing, and most distributions do not include these drivers.

Although the installation is more complex, as you need to build from the source code, ENBD does support error handling, automatic re-connect if a connection fails, multiple channels, and more.

Going Wireless

DNBD [6] (Distributed Network Block Device) is specially designed for diskless operations on wireless networks. In this scenario, clients associate with an access point and share frequencies. Only one client will transmit at any time, a fact that is complicated by the restricted bandwidth of 54 Mbps with today's IEEE standards 802.11b/a/g. As all the clients share the network medium, the available data rate is further reduced by each additional device that attempts to boot simultaneously.

DNBD attempts to minimize the volume of data as much as possible, while at the same time leveraging the abilities of the wireless network. To achieve this aim, DNBD does without locking mechanisms and only supports read access. Additionally, DNBD data traffic is visible to all the clients attached to the network (Figure 6), and the clients cache every single block the server transmits, no matter which client it was intended for. After all, it is likely that the other clients will need the block sometime.

Although the shared medium in a wireless network principally allows clients to eavesdrop on other clients' conversations, not all wireless cards support promiscuous or monitoring mode. To overcome this, DNBD uses IP multicasting to address a group of clients simultaneously. To allow this to happen, DNBD relies on the connectionless UDP protocol. DNBD handles lost packets and other communication problems itself. It uses IP multicast addresses in the 224.0.0.0/4 network range, although 239.0.0.0/8 is recommended for your experiments.

Before a DNBD client can access a network block device, it first attempts to discover a server by sending a special packet to a defined multicast address. The server response includes information about the existing block device, which the clients can use for configuration purposes. Clients can then request blocks. DNBD servers can be replicated in high availability scenarios. Clients discover servers automatically; they continually collect statistical data to be able to choose the server with the lowest response time.

Meaningful caching always assumes locality properties. Clients need the ability to access the same blocks for a limited period of time to benefit from this approach. These conditions apply when a large number of diskless clients boot simultaneously on a wireless network. But it also works for multimedia content: multiple clients can play a DVD over a WLAN, and the cache will still work if replaying is time-shifted.

The DNBD project page includes an installation Howto that describes installing DNBD via Subversion. You need to have and configure the kernel sources or headers: you may need to modify the path to the kernel in *kernel/Makefile*. Just like other network block devices, DNBD has some trouble with various combinations of I/O schedulers with filesystems, especially in diskless operations.

Although PXE Linux (Pre-boot Execution Environment) [8] and Etherboot [9] were introduced to support booting PCs over Ethernet, a standard for booting over a wireless LAN has not been forthcoming. Instead of using the ether to send the kernel and the ramdisk to a machine, USB sticks or compact flash cards attached to the IDE port are useful kernel carriers. And let's not forget that *init* on the Initram-FS is assigned the additional task of setting the WLAN parameters.

Well Blocked

NFS causes considerable overhead, a big hindrance if you are dealing with small files. More specifically, the typical runlevel, system setup, and admin scripts on a Linux system make life difficult for NFS. And the situation is aggravated by the new tricks that modern distributions get up to with filesystem monitors and other components generating a steady flow of NFS data.

Network block devices work better with the internal kernel cache, and an inactive system will thus generate almost no network traffic. On opening a file, Linux only reads the block once to ascertain the permissions, owner, or last changed date, and to pick up a file handle. NFS needs a whole bunch of RPC calls to do this.

NFS operations can also be optimized. Letting the typical setup and configuration routines run in an initial ramdisk will reduce the gap between NFS and the block device. Using the compact SquashFS [7] is an interesting option: in read-only scenarios, precompressing the filesystem can reduce network traffic and put the block caches to more efficient use.

Using Union-FS or Cowloop has a marked effect on performance. The latter is restricted to use with a block device that has a writable filesystem. Union-FS is useful in any scenario and will also help reduce traffic. Additionally, Union-FS helps to reduce the load on the expensive Temp-FS in RAM, especially with regard to typically small configuration files.

High availability is difficult to achieve with NFS. DNBD specializes in high availability solutions, proving far more accomplished at switching transparently – from the client's point of view – to alternative servers. System administrators can even dynamically add and remove servers from the network without the users even noticing the change, providing an elegant workaround for a potentially disastrous single point of failure if a server crashes.

Specialist

Linux network block devices were not designed to send the traditional network filesystem into retirement, but block devices definitely provide an interesting option for scenarios that feature multiple diskless clients. The techniques described in this article will help you get started with providing fast and reliable file service to the diskless clients on your own network. ■

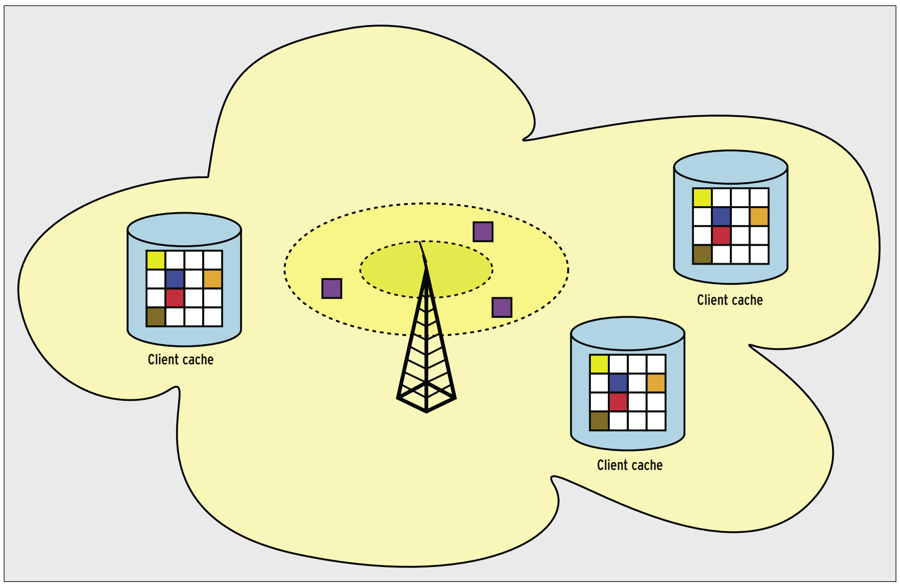


Figure 6: Distributed NBD is best-suited to wireless networks. When a computer requests a bloc, the server multicasts the block to all clients. This saves time and bandwidth if the next client needs the same block.

INFO

[1] Cowloop: *http://www.atconsultancy.nl/cowloop/*

[2] NBD (Network Block Device): *http://nbd.sourceforge.net*

[3] ENBD (Enhanced Network Block Device): *http://www.it.uc3m.es/ptb/nbd/*

[4] GNBD (Global Network Block Device): *http://sources.redhat.com/cluster/gnbd/*

[5] ANBD (Another Network Block Device): *http://www.aros.net/~ldl/anbd/*

[6] DNBD (Distributed Network Block Device): *http://lp-srv02a.ruf.uni-freiburg.de/trac/dnbd/*

[7] Squash-FS: *http://squashfs.sourceforge.net*

[8] PXE-Linux: *http://syslinux.zytor.com/pxe.php*

[9] Etherboot: *http://etherboot.sourceforge.net*

THE AUTHORS

Thorsten Zitterell works for the Operating System Department at the University of Freiburg where he researches into realtime operating systems in embedded microsystems.

Dirk von Suchodoletz is an assistant at the Department of Communication Systems and constantly on the lookout for intelligent designs to support diskless clients on Linux.