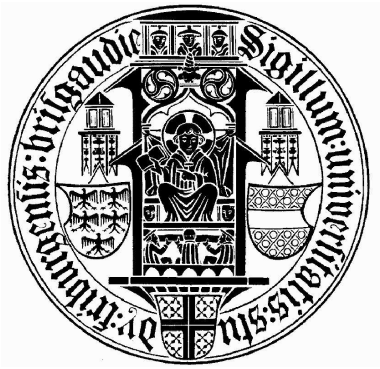
Development of a Distributed Network Block Device for Wireless Clients

Diploma Thesis

by

Thorsten Zitterell



Albert-Ludwigs-University Freiburg

Faculty of Applied Sciences

Chair of Communication Systems

Prof. Dr. Gerhard Schneider
Advisor: Dirk von Suchodoletz

January, 2006

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und sämtliche Stellen, die wörtlich oder sinngemäß aus veröffentlichter oder unveröffentlichter Literatur entnommen wurden, als solche kenntlich gemacht habe. Außerdem erkläre ich, dass die Arbeit nicht – auch nicht auszugsweise – bereits für eine andere Prüfung angefertigt wurde.

Thorsten Zitterell

Freiburg, den 12. Januar 2006

Contents

Acknowledgements

My thank is addressed to all people, who supported me during this diploma thesis. First, I want to thank my supervisor Dirk von Suchodoletz for providing me with support, assistance and suggestions during this thesis, and I want to thank Prof. Dr. Gerhard Schneider for his kind engagement and taking the responsibility for this work. I want to thank Marion Reitter sincerely for her patience and moral support, and Zeno Gantner and Sören Roerden for valuable discussions. Finally and foremost, I want to express my gratitude to my family for their support throughout my whole course of studies.

Abstract

Central organization of data storage has many advantages regarding maintainability. Diskless workstations tie up to this concept, but their operation makes high demands on the underlying network infrastructure. Radio networks can not compete with wired infrastructures in regard to bandwidth, reliability and scalability due to a lossy and shared communication medium. However, diskless clients offer a special environment, because they usually are homogenous systems – their basic system does not change over a long period of time.

In this thesis, these characteristics are used for the development of a distributed read-only network block device (DNBD) for wireless clients which minimizes the network traffic by one-to-many semantics and caching of data exchange by other nodes. Additionally, it integrates flow control mechanisms for unreliable networks and thus is, to our knowledge, more suitable than present solutions.

1. Introduction

1.1. Scope and Motivation

Central organization of data has many advantages regarding network environments. Many companies and institutions provide central servers for a reliable storage of user files or administer databases for personal, mercantile and other internal information purposes. Further reasons, in addition to data safety and simplified sharing, are high availability and lower costs for the extension of capacities (e.g. disk space) or backup creation [9]. Diskless workstations tie up to the approach of central data management. Such workstations obtain their operating system over the network and do usually not possess a local hard disk. Less effort for maintenance by central software updates and configuration are reasons for their application. Minimal downtime in case of hardware defects is reached, as the diskless system is then simply replaced. Of course, central data organization also has some drawbacks because it represents a *single point of failure*[1] and involves problems in scalability; however, some problems can be solved by replication, for example.

In this thesis, we consider wireless, diskless workstations – or briefly *wireless clients* as we will call them henceforth. Wireless technologies can not come up with the bandwidth and reliability of wired networks. Widely spread technologies like the IEEE 802.11 b/a/g[2] standards offer maximum data rates of 54 Mbit/s, although some vendors have increased the bandwidth to 108 Mbit/s by proprietary extensions. Other wireless communication methods are much slower (e.g. bluetooth, UMTS etc.) or depend on expensive and unstandardized hardware. In practice, all wireless technologies have another problem in common: The available bandwidth is limited by the number of devices using the same frequency and medium. As a result, computer pools with wireless clients have major drawbacks compared to wired configurations.

These problems give rise to the question – *"how can we adapt present solutions and take advantage of characteristics in a diskless and wireless system to provide a highly-optimized network storage solution for such an environment?"* In order

[1] Single points of failure are components which cause, in case of a failure, a complete breakdown of a system.

[2] IEEE 802.11 is a wireless standard developed by the Institute of Electrical and Electronics Engineers.

to discuss this question, we have to consider several aspects of such a distributed system with shared resources [6]:

- **Heterogeneity:** The used communication protocol must be adaptable to a variety of wireless networks and hardware.
- **Openness:** Used interfaces, protocols etc. have to be documented to allow further extensions.
- **Security:** Requirements for a secure operation must be defined.
- **Failure handling:** Down-times should be minimized in case of single failures.
- **Concurrency:** Concurrent access to resources must be safe and should be avoided for better scalability.
- **Scalability:** The used algorithms, data structures, protocols etc. should avoid performance bottlenecks and costs for further clients should be minimized.

1.2. Contributions

In this work, we will consider these aspects and present the *Distributed Network Block Device*[3] (DNBD), which can be seen as a *virtual* drop-in replacement for a local hard disk in wireless workstations. As DNBD operates in wireless environments, communication between servers and clients is optimized for minimized traffic and flow control mechanisms are implemented to perform better in face of limited bandwidth and lossy communication. Complexity and costs in communication are decreased, as write semantics are shifted from the server to clients. The possibility of replicated servers helps to keep down-times low and provide load balancing. We propose one-to-many semantics in communication by multicast and pre-caching techniques, in order to avoid redundant data sending in a shared medium.

The resulting implementation runs under the Linux operating system. We intend to release our Distributed Network Block Device as open source software.

1.3. Structure

After this introduction, we will present related work and projects, which provide similar functionality in regard to data storage in networks. The following chapter 3

[3]The naming *block device* refers to the abstraction of devices in operating systems which exchange data in blocks like hard disks, CD-ROMs etc.

deals with the design of our distributed network block device. We will define requirements and introduce main components of such a system. Implementational aspects are discussed in chapter 4, considering used interfaces and data structures provided by the operating system. Experimental results of our implementation are presented in chapter 5. Chapter 6 summarizes and concludes this thesis.

2. Related Work

Until now there have been several implementations of a network block device under Linux. The projects discussed below have many characteristics in common and are not adapted to a specific environment like DNBD for Wireless Clients. Internet links to these projects are listed in appendix C.

2.1. Network Block Device (NBD)

The Network Block Device by Pavel Machek is the only driver of this group included in the official Linux kernel since version 2.1.101. There have since been some patches to support swapping over NBD, SMP-safeness and 64-bit architectures.

Its characteristics are described in the documentation tree of the kernel sources[1] as follows: *With this compiled in the kernel (or as a module), Linux can use a remote server as one of its block devices. So every time the client computer wants to read, e.g., /dev/nb0, it sends a request over TCP to the server, which will reply with the data read. This can be used for stations with low disk space (or even diskless - if you boot from floppy) to borrow disk space from another computer.* The usage of NBD is quite simple, as most Linux distributions already include the necessary kernel module and user space applications. Once the server is started with an arbitrary block device (e.g. the partition `/dev/hda7`) and a free TCP port (e.g. 7777) on which it should listen, the client can import that device as `/dev/nb0` and access the partition locally:

```
root@server # nbd-server 7777 /dev/hda7
root@client # nbd-client server 7777 /dev/nb0
```

Now the remote block device can be equipped with a common file system (if not already done) and mounted in the directory tree. All following projects use the same principles.

[1] http://www.kernel.org/

2.2. Enhanced Network Block Device (ENBD)

The Enhanced Network Block Device (ENBD) by Peter T. Breuer arose from a backport of the native NBD driver mentioned above and was developed to provide failure-mode and redundancy features [17]. The most important changes with respect to the NBD driver in the kernel are as follows:

- multiple communication channels, giving the opportunity to improve throughput, especially on multi CPU platforms
- failure handling by automatic restart, reconnect and recovery of a session
- support for partitioning NBD devices and removable media

These enhancements make it production stable and numerous articles deal with its application (e.g. [3]). An interesting appliance is to use it in combination with software RAID, so that a local and remote disk can be kept synchronized.

2.3. Another Network Block Device (ANBD)

With *Another Network Block Device*, Louis D. Langholtz has written a NBD compatible server and client implementation with multi-threading support. Its main goal was – in his own words – to implement *an elegant, maintainable, robust, speed-wise optimal, large file supporting, improved implementation of the NBD tools, architectured with concurrent handling of newer NBD style protocols in mind.* Just as NBD and ENBD it uses TCP connections to transfer data blocks between server and client.

In fact, some constrictions of the original NBD have been revised. These are amongst others flexible configuration, some extensions (e.g. copy-on-write mode), better monitoring and more tolerant handling of temporary network failures. Concerning transfer speed issues, its performance is equivalent to the original driver. As ANBD is multi-threaded, some concepts, concerning threads and queues, have been adopted by the author, as we will describe later.

2.4. Global Network Block Device (GNBD)

GNBD is part of the *Global File System* (GFS) project, a file system especially for clusters, which uses either a central locking server or a distributed locking manager [10]. It was originally developed at the University of Minnesota. Today, it is maintained by Red Hat, Inc. and available to everyone under an open source license. The *Global Network Block Device* works, in principle, as the network block devices mentioned above, whereas GFS includes a lock module which coordinates I/O access. Thus, several clients can access the same file without corrupting the underlying structure of the block device. However, GFS does not depend on GNBD but can use any block oriented devices like iSCSI, Fibre Channel etc. Thus, it should work with all network block devices which allow multiple client access.

2.5. Remarks

Here, the presented block devices are general purpose or developed for specific file systems (e.g. GFS). Some include reasonable extensions like further safety mechanisms and are a cheap alternative to hardware solutions usually based on Fibre Channel or iSCSI.

However, all implementations mentioned above use TCP communication. We will see in section 3.2.3 that TCP will not fit the requirements for the design of our block device in wireless environments, as its implementation does not provide one-to-many semantics in communication and has problems with intermittent and lossy data exchange in wireless environments.

3. Design

In this chapter, we discuss the design of the *Distributed Network Block Device* (DNBD). At first, we define the requirements for our implementation. The system design is covered in section 3.2 which explains the network storage model, communication, replication, traffic control and caching. The resulting components of the server-client architecture are presented in section 3.3. Finally, limitations of DNBD are discussed in section 4.4.

3.1. Requirements

The *Distributed Network Block Device* is intended for the use with wireless clients – although it is not limited to such an environment. DNBD avoids unessential data exchange for wireless clients operation by simplifying the data exchange protocol and integrating further enhancements. We presume that wireless clients are workstations with a homogeneous software configuration including the operating system and common user space applications (e.g. browsers and mail clients, office and multimedia applications). In practice, this configuration stays unchanged over weeks or even months.[1] Minor changes of the system like temporary file creation should not affect the original system on the server. Therefore, the system on the server is exported read-only and write access can be handled locally on the client by copy-on-write methods. As the system is the same for all participating nodes redundant data exchange can occur if different clients access the same information. A remedy for this is pre-caching the data exchange of other nodes, which has been implemented for DNBD. This approach is evaluated in chapter 5.

This chapter attends design aspects of a network block device for wireless clients. The main requirements of DNBD are summarized as follows:

- The objective of DNBD is to develop a highly scalable and robust network data storage solution, which minimizes network traffic and bandwidth utilization.
- DNBD's communication protocol has to cope with intermittent and lossy data exchange in unreliable wireless networks.

[1] Only security updates or new distribution releases are usually reasons for system changes.

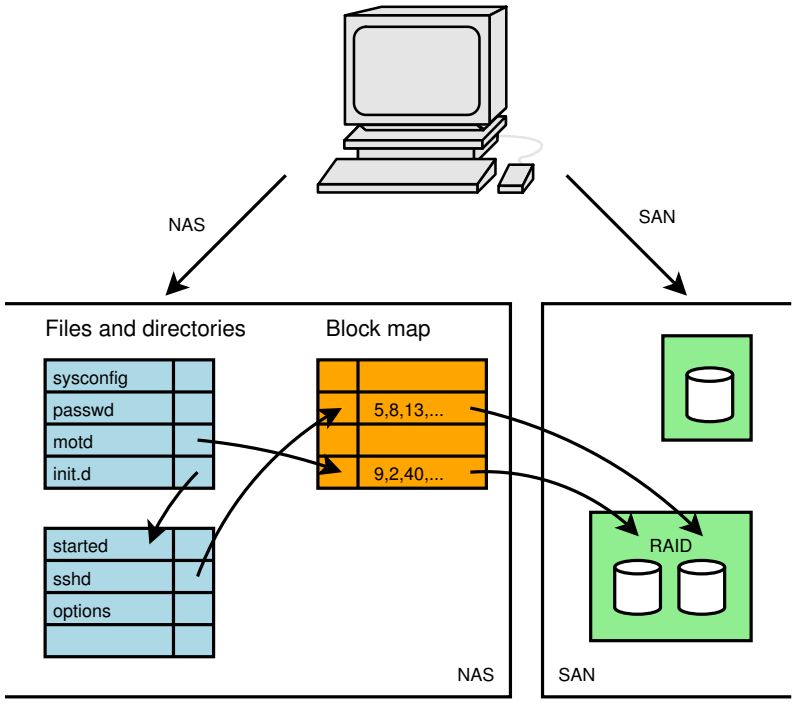


Figure 3.1.: A client can either access files over a network file system (NAS) or directly access data in the form of blocks. In the latter case, the client uses a local file system based on a remote disk.

- One-to-many semantics in network communication are used to address a group of client nodes and to make pre-caching methods possible.

- DNBD operates read-only on the server as write semantics can be shifted from the server to the clients in order to suppress expensive locking mechanisms.

3.2. System Design

3.2.1. Network Storage

Today, there are two basic approaches to share data in a network. The first one is *network attached storage* (NAS), which can be seen as a file server. *Storage area networks* (SAN), on the other hand, are systems where storage resources are exchanged in data blocks. For robustness, less complexity and higher efficiency, DNBD uses the SAN approach, as we will discuss later.

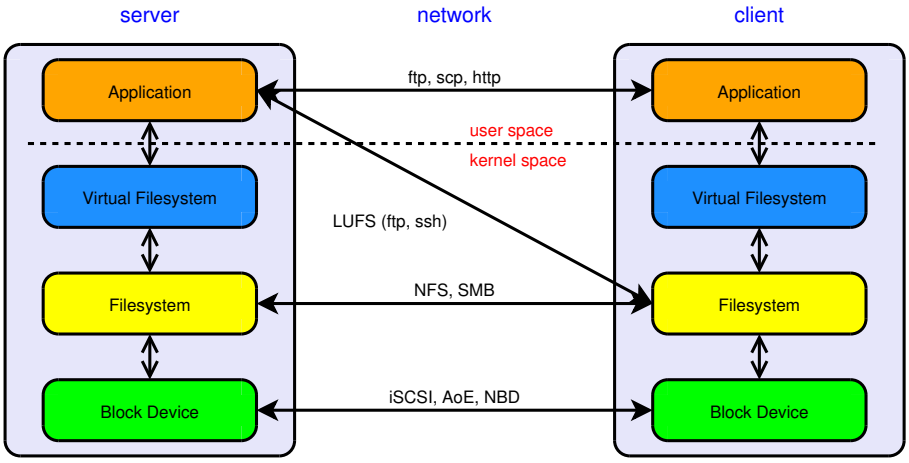


Figure 3.2.: Data can be exchanged on different levels of the network protocol. Some protocols (e.g. iSCSI) are based on blocks and others operate directly with files. High-level protocols like *file transfer protocol* (FTP) are usually implemented on application level at the client but the *Linux Userland Filesystem* (LUFS) makes it possible to integrate the files on file system level.

SAN and NAS

There can be several advantages to shifting data storage from computers to central servers. These are better data safety, availability and sharing, reduced waste of resources by merging unused capacities and flexible management. The disadvantages of such solutions are a higher degree of complexity and the requirement that all components function correctly. In contrast to self-contained systems, network storage solutions have to take into account security issues, as malicious use of the network can compromise privacy and data integrity. As mentioned above, NAS systems provide file system functionality to clients, whereas SAN systems offer a memory-like interface based on data blocks [9].

Storage area networks and network attached storage integrate at different levels in an operating system. The exchange of blocks within a SAN is, in principle, the same as accessing the blocks of a local hard disk. In fact, the operating system on the client leaves this task to an underlying device driver which offers the same interface and routines. Here, the device driver encapsulates the request in a specific network packet and sends it to the server, which handles the request and sends an appropriate success or failure message or a data block back. As SAN is usually established for one-to-one semantics (one server, one client), locking methods for shared access to a block are not considered. The advantage of SAN is the spatial separation of data storage and data processing, usually combined with the integrity, fault-tolerance and throughput of a RAID[2] system.

[2]Redundant Array of Independent Disks

The operation of NAS is more complex. Here, the data storage is implemented as a file system on client side, and consequently, it has to provide functions for opening, reading and writing, seeking, attribute handling and much more. In contrast to SAN, *many* clients share files on *one* server and the server has to regulate concurrent access to resources. The common file system semantics of an operating system are extremely difficult to implement in a distributed system: In a local system, it is no problem for two processes to write to a file at different positions – however, in a distributed system it has to be ensured that all clients (holding this file open) are aware of these changes. As this would lead to scalability problems, most network file systems use weaker semantics; e.g. files are changed separately for each client and written back when they are closed. Of course this can break consistency of files, but makes it possible to use further optimizations like caching. NAS itself can be based on SAN – or in other words – the underlying data of a file system provided by a server can be placed on remote media accessible over the network.

DNBD as a SAN solution

The Distributed Network Block Device covered in this work is based on the SAN approach and therefore uses block transfers between server and client. The reasons for a block device approach is primarily the reduced complexity, both on the server for data access and on the client for caching. A file system implementation with similar characteristics (cp. section 3.1) entails much higher costs on the server by opening, splitting, transferring and closing the file. With each access, the operating system verifies if the server application may access the file and the client has to signal when the file is completely transfered (or is not needed any more), so that the server can close it. The file system driver on the client has to implement most of the functions of the *Virtual File System* interface and special treatment of large files which do not fit into main memory. Cache organization is more expensive as a key for the filename *and* the position of the cached page should be stored: Assuming a client wants to cache files which another one has requested and it misses only one block – the collected information would be useless if files were cached as a whole.

The complete process of file segmentation to a suitable form ready to transfer over the network and the possibility of caching is less expensive, if we approach data exchange one level lower in the network storage model (figure 3.2). A block device located on the server contains a file system which *is* already segmentated and suitable to be transferred in blocks. These blocks can easily be cached with their position in the block device as key.

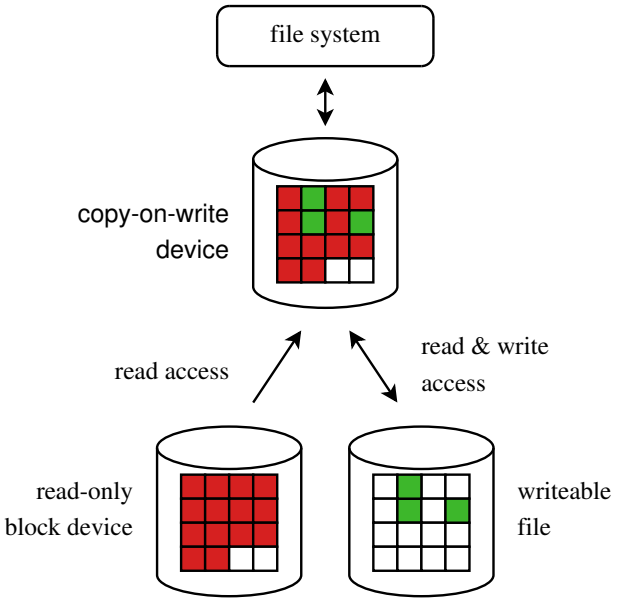


Figure 3.3.: Copy-on-write semantics make a block device virtually writable. Modified blocks are written to a file and used instead of the original one from now an.

3.2.2. Write Semantics

In a wireless clients environment, the server provides data storage suitable for the clients to boot the system (including start-up of daemons, helper programs etc.) and provides necessary workstation applications, such as a window manager and common user programs. There are only a few files and directories which are expected to be writable by a client's system. Typical examples are files which temporarily store runtime information like network information (`/etc/resolv.conf`), process identifiers (`/var/run/syslog.pid`), etc. All of these are specific to *each* client and sharing them with other clients is not necessary. Therefore, the Distributed Network Block Device does not provide write access on the server and each client locally takes care of write semantics in its own system environment. Additionally, server and client nodes need not be synchronized and locking mechanisms for concurrent write access to resources can be abandoned, which reduces network traffic to mandatory block transfers from the server to a client – the main goal of DNBD's design.

There are several approaches to adapt a read-only system in order to gain write access. A trivial procedure is to copy those files of a directory, which have to be changed, to a writable file system (e.g. on a ramdisk) and replace the original directory with it. More elegant ways use copy-on-write semantics either on the block or file system layer of the operating system. The principle of both solutions is to transparently integrate with the operating system and redirect write access to another device.

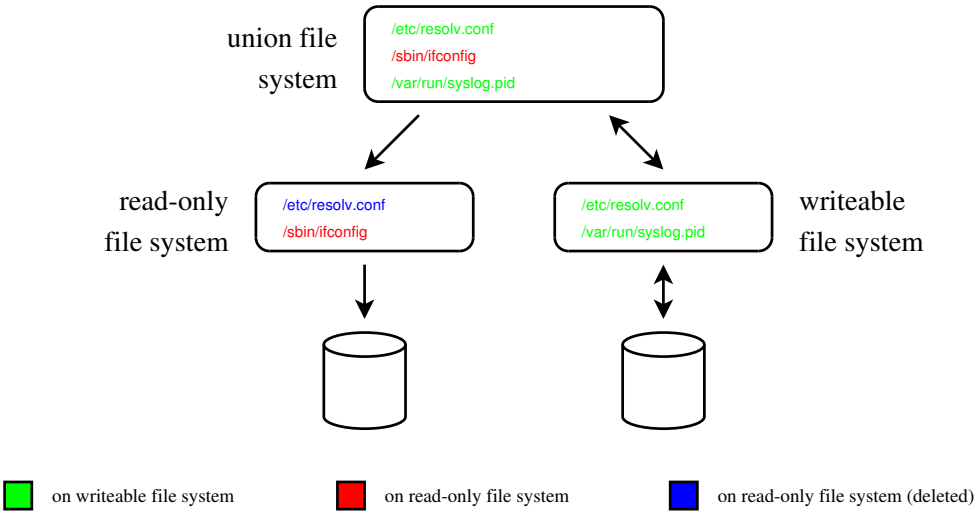


Figure 3.4.: Files that are opened for writing are copied to a writable file system whereas the original file is hidden. The union file system transparently merges contents of both underlying file system.

An implementation which uses the block level approach is `cowloop` (appendix C). This driver takes a read-only block device (e.g. from DNBD) and combines it with a writable block device: Whenever a block has to be changed, it is copied from the read-only to the writable device and then altered (figure 3.3). The internal design of the file system approach – a so called union file system – is far more complex, as many special cases have to be considered,[3] but the principle is the same: In contrast to cowloop, files – and not blocks – are copied before they are changed (figure 3.4). It does not matter wether a block device or file system level approach is adopted for DNBD: cowloop can be used directly on top of DNBD, whereas a union file system can utilize a file system with which DNBD is equipped.

3.2.3. Communication

Whenever data storage is shared over a network, we can distinguish between one-to-one or one-to-many relationships. The latter is mostly the case in networks of workstations, where many clients share the same volume on a server. In general the performance is limited by the bandwidth of the underlying network, because peer-to-peer connections are established on the transport layer of the underlying network between each client and the server. Caching and replication are ways of dealing with these difficulties, but usually entail an overhead for locking and keeping cache consistency.

[3]The file system approach has been discussed in a former work by the author [16] and is only outlined here.

In wireless environments a shared medium for communication is predetermined and the limited number of communication channels has to be shared among the clients. However, wireless clients can overhear the communication of others which can be utilized for pre-caching.

The DNBD protocol

In a typical environment, at least one server and an unspecific number of clients are present. An appropriate communication protocol has to be defined, so that clients can request a data block from a server – and the server replies with the requested information. Thus, at least an identifier for the requested block is needed – or alternatively a position pointer which tells the server *where* data has to be read from the underlying block device. Additionally, the server must know how many data units have to be transferred. This could indeed be a fixed value, however, Linux internally supports dynamic block sizes for a block device, which can change for several reasons during operation.[4] If this fixed size is smaller than the size expected by the client, it can not process this data fragment because the block I/O size described later in section 4.1.3 has a lower limit. Basically, the information *where* (position) and *how much* (amount) has to be read are sufficient for block transfers, but for session management and probably further extensions, four more fields are adopted.

Whenever a client initiates a new session, it does not know any details about the block device provided by the server. One detail could be if the the media is removable or not, but the most important one for proper operation is the *size* of the block device. With this information the clients know if they are trying to access blocks which are beyond the end of device. Thus, the clients can not only send requests for block transfers, but also request properties of the block device itself; for this, a field is used as a command identifier telling the server the purpose of a request.

As we will see in section 3.2.5, DNBD will use a decentralized algorithm for destination designation. In this manner, the clients will decide which server should reply. Here, a server is addressed by another identifier field. A unique identifier is prescribed when each server is started. At session startup and periodically during a session, the clients ask for available servers and dynamically determine which server would give the best throughput. In this context, another field is used as time stamp information in order to update round-trip times.

Finally, a sixth field with a so-called *magic number* is established to avoid interference with other protocols. The magic number is a constant and assigns a packet to

[4] When a file system is mounted the file system implementation can change the block device size to another value. Block sizes can also be changed by user space tools, e.g. `blockdev --setbsz <size> <device>`.

our DNBD protocol. If the server or client application of DNBD receives a packet which does not match a specific packet size or the magic number, they can drop the packet; otherwise it is assumed that the packet should be processed. However, if other authentication methods or a *virtual private network* were to be established this field could be omitted.

Similar to the request, the reply packet contains again the magic number, the unique identifier of the server and the command field and position. The other fields depend on the command identifier. When a session is started, the capacity of the device and an initial block size is submitted. For block I/O requests the position field is added and the block is appended to the packet. An overview of the protocol is given in appendix B.

IP Integration

The DNBD protocol is the middleware protocol and a transport protocol for network communication is needed. The network block devices discussed in section 2 use TCP/IP for this, but DNBD is based on connectionless UDP communication.

TCP can provide many advantages in comparison to UDP. It is connection-based and the operating system is responsible for reliable data exchange. For this, the TCP header includes a *sequence* and *acknowledgement number* in order to preserve the packet order and to detect packet duplication and loss. Additionally, a *window size* field tells the connected nodes if the remote station still has free buffers and is used for flow control. However, all these fields are overhead to the DNBD protocol as traffic control is done in a different way. Sequence numbers can be omitted as DNBD uses position numbers (section 3.2.3) for requests and replies. Everytime a request is sent, it is requeued as an *outstanding* request; whenever a reply is received, it is checked if there is a corresponding outstanding request. Thus, the order of received replies is not relevant. Packet loss detection and other flow related control is handled by a timing mechanism which will be discussed in section 3.2.5.

A further reason not to use TCP in a wireless environment is that it does not really perform well in such intermittent and delayed networks: Traffic flow, timeout behavior and other characteristics of the TCP stack seem to drastically suffer from packet loss and jitter. Numerous papers deal with these problems and propose new protocols, e.g. *Wireless TCP* (WTCP) [7].

In regard to a wireless environment, a shared medium for data transmission is predetermined, which is usually a drawback in communication because only one node can transmit at a particular time. However, DNBD's effort is to make the best of it by overhearing other nodes' packet exchange. A data block which was formerly requested by another node could be cached and accessed later without asking the

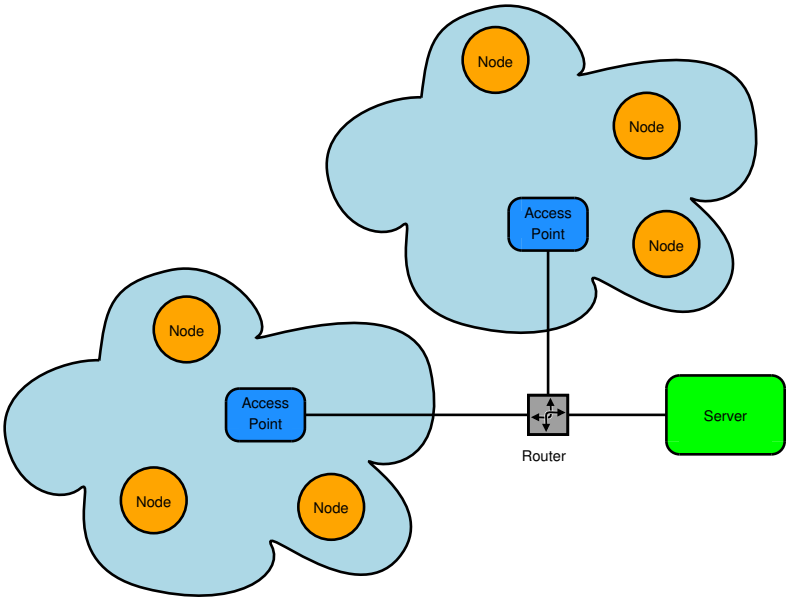


Figure 3.5.: The router in this diagram is aware of adjacent multicast networks. For Linux, multicast capabilities are part of the kernel. When applications bind or unbind a multicast address, the kernel automatically sends messages to inform multicast routers about new or leaving group members.

server again. We will discuss this caching method in section 3.2.6. Concerning communication, we have not specified *how* datagrams can be intercepted. One method is the so-called promiscuous or monitor mode of a network card which will pass all network traffic on to the operating system – even if it is not the intended recipient. However, this method usually entails technical problems with many wireless network cards. Some cards lack firmware support for such a monitor mode and others need specific commands to enable it. Thus, a general approach would be difficult. Another possibility is to address a group of nodes by multicast techniques and DNBD uses such a method with the help of IP multicast. In contrast to the promiscuous mode approach, IP multicast works across subnet borders with the help of multicast routers (figure 3.5) and is independent of the underlying network device. IP multicast in Linux only works for UDP and is not implemented for TCP – however, this is not a constriction to DNDB as it is satisfied with connectionless communication.

3.2.4. Replication

In general, *replication* is provision of the same data at different locations. Besides higher availability of data and fault tolerance it gives a better possibility of load balancing. In computer science, replication is used in many applications and in various ways. A common example is the *domain name system* (DNS) of the Internet, which

does name-to-attribute mapping and achieves high performance and fault tolerance even when parts of the server system fail. Replication is detailedly discussed in [6, Ch. 14].

The question of service availability is significant for distributed systems. A quantitative estimate of availability in case of server failures and network partitions can be determined in the following way. If n servers have an independent probability p of becoming unreachable, then the probability that all servers fail is p^n, or respectively $1 - p^n$, that at least one server is available. Thus, the availability of a single server with a 4% probability of failure is 96%. Three independent servers with the same characteristics would have an availability of $1 - 0.04^3 = 99.9936\%$. In practice, there are more criteria and dependencies among the participating components, but the example shows that replication plays a major role in the reliability of a system.

Replicated servers do not only provide higher availability for their clients, but can also be used to share workload between them. A problem is how failed servers can be determined to avoid unnecessary queries and how requests can be distributed. Approaches to solving such problems will be discussed in the next section.

3.2.5. Traffic Control

Destination Designation

In case of multiple or replicated servers, a smart destination designation scheme is needed. In contrast to many applications which use a round-robin method to balance between the servers, we adapt an algorithm from *admission control for anycast flows* [12] to find the nearest servers. *Nearest* in the sense that end-to-end delay between request and reply is minimal. For better scalability, a distributed algorithm is proposed where decisions are made by the individual nodes.

To that end, each client keeps a list of possible servers. Servers are searched by a client at the beginning of a session. The number of servers is S. Each server can be given a weight $W_1, \ldots, W_S$ which represents the probability that this destination will be queried. However, the weight must fulfill the condition

$$\sum_{i=1}^{S} W_i = 1.$$

Clients and servers in DNBD follow a request-reply protocol. Thus, a *round-trip time* (RTT_i) can be proactively determined between sending a request and receiving

a reply, and an exponential average ($SRTT_i$) is acquired for each server $i = 1, \ldots, S$ and request n:

$$SRTT_i(n) = \beta SRTT_i(n-1) + (1-\beta)RTT_i(n)$$

$RTT_i(n)$ is the current measurement of the round-trip time. The value β should be chosen high enough to give single measurements less significance ($\beta > 0.9$ in practice).

Now, a weight for each server can be assigned which is inversely proportional to the exponential average of the corresponding round-trip time:

$$W_i \sim \frac{1}{SRTT_i(n)}.$$

Finally, the normalized weights are defined as

$$W_i = \frac{1/SRTT_i(n)}{\sum_{j=1}^{S} 1/SRTT_j(n)} \qquad (i = 1, \ldots, n).$$

In DNBD, these weights are periodically calculated and servers are chosen according to their weight or probability, respectively.

Lossy Links and Stalled Servers

The round-trip time is also used to discover packet losses. A request which is not answered within a multiple of this time is retransmitted. The factor, used in this context, should not be chosen too low, in order to avoid retransmit bursts, and not too high, for responsiveness.

For production use, it is advisable that a network storage solution be able to handle single server failures. For DNBD, several replicated servers enormously increase availability, as we have seen in section 3.2.4. As DNBD uses a distributed algorithm for destination designation, a method for detection of stalled servers has to be considered. The round-trip time algorithm introduced above is insufficient, because stalled servers will not reply to any requests and the values are not adapted any more.

Therefore, the last point in time when a packet was received is stored for each server. If the failure duration exceeds a given time-out interval, the server will be disabled. A periodically activated session thread requests heartbeat packets of all available servers, so that new or formerly disabled servers are (re)activated.

3.2.6. Caching

At the beginning of this chapter we determined some drawbacks of wireless networks. In order to optimize DNBD for minimal network traffic, we suggest a pre-caching mechanism to reduce redundant data exchange.

Wireless environments usually operate in *ad-hoc* or *infrastructure* mode. In ad-hoc mode all wireless clients can directly communicate with each other (as long as they share the same protocol and frequency). However, in most environments – no matter if in homes, companies or universities – access points (as bridges to wired networks) are installed to provide central connection points. IEEE 802.11 is a common standard used for Wireless LANs and encompasses the physical and data link layer of the well-known OSI model. The maximum raw data rate achieved is 11 Mbit/s for IEEE 802.11 b, but in practice maximum throughput is about 6 Mbit/s for TCP and 7 Mbit/s for UDP. Depending on the wireless system which has to be booted over network, many megabytes have to be transferred until a system is usable. A typical system boot with a default SuSE 10.0 installation will transfer between 80 MB and 150 MB depending on the used network data storage. In the case that 100 MB have to be transmitted with an effective rate of 0.5 MB/s, it will approximately take 200 seconds until the system is ready. However, if n clients boot at the same time and the network can only be used by one client at a time, we obtain a $O(n)$ complexity and, for example, boot up of 18 clients will last one hour.[5]

DNBD tries to minimize the amount of traffic by caching server replies to other clients' requests. However, the use of a cache is only appropriate if a locality of reference can be exploited. In the case of simultaneously booting systems we increase the temporal locality, as all systems need the same blocks from the server. Additionally, many system services are activated at a specific time and can produce concurrent network traffic: A typical representative is `locate` which scans the whole file system and creates an index to make searching of files faster. Even human interaction with the system can lead to bursts – especially when wireless clients are used in an educational environment and an instructor tells the listeners to perform specific steps with their computer.

Cache Organization

In section 3.2.3 we described the DNBD protocol and how block requests are transferred between server and clients. The unique identifier of a block is its position or sector number, respectively, within the block device and provides a key for cache operations. For flexibility, cached blocks are kept in a file which can be either located in memory (e.g. ramdisk) or on any other file system carrier and has a specific size.

[5] In this simplification, it is assumed that the network is the limiting part.

Additionally, we assume that the size of the cache is usually smaller than the size of the block device – and not all blocks of the block device can be cached locally on the client. These specifications define the framework for the cache, however, the choice of appropriate data structures and algorithms for cache organization and operations requires a closer look at DNBD and where searching, insertion and removal of cache entries can occur.

- **Sending requests:**
  Before a block is requested over the network, it is checked if the block is available in cache. If it is existent, the corresponding block can be copied to the given address in memory and is completed. Otherwise, DNBD sends the request to the server.

- **Receiving replies:**
  Whenever a reply to another clients' request[6] is received, DNBD searches the block. If the block is not found, it is added to the cache; however, a replacement policy is needed, if there is no space left.

Therefore, searching with the block number as a key is the most frequent operation in our cache, as it is needed when sending requests and receiving replies. Insertion and replacement of cached blocks depends on the number of replies requested by other client nodes. As usual, the mapping between cached blocks (indexed by their position on the block device) and their position in the cache is done by an associative array.

Efficient data structures for associative arrays are hash tables and self-balancing binary search trees (BST). Both representations have their advantages and disadvantages. Hash tables can offer an average lookup time of $O(1)$, but in worst case it is $O(n)$ whereas self-balancing binary search trees can search, insert and delete in $O(\log n)$ time. For DNBD, the search tree approach has been chosen. Although the worst case would rarely arise with an appropriate hash function, its occurrence could rapidly decrease cache performance.[7]

Popular data structures for implementing self-balancing BST are, for example, AVL, splay and red-black trees [5] [15]. Characteristics of BST mostly differ in average and guaranteed worst-case running time for the typical insert, delete, search and rebalance operations. For DNBD, the red-black tree approach has been chosen to organize the cache, as it offers a good worst-case running-time with a overall $O(\log n)$ complexity for those operations. Other BST implementations would have similar characteristics and could probably have a slightly better performance for this application. For a start, a *fast* solution is satisfactory. An in-depth evaluation

[6] Replies to own requests are not cached as this is already done by the kernel. (cp. section 4.1.3)

[7] The choice of a hash function would also depend on the file system used on top of DNBD as any file system organizes its data in a different way.

of different data structures for caching is deferred for future work, as further analysis, mainly in interaction with an internal behavior of the operating system (e.g. buffers), goes beyond the scope of this work.

Red-black Trees and Replacement Policy

Primarily, red-black trees have been described by Rudolf Bayer in 1972 [1].[8] As the name implies, each node of the tree has an additional property, an associated color which is either red or black. Moreover, the tree has to fit some conditions in order to guarantee that it stays balanced to a certain degree:

1. Each node is either red or black.
2. The root of the tree is always black.
3. Each leaf node is black.
4. A red node must not have a red child.
5. All paths from an arbitrary node to its leaves contains the same number of black nodes.

With these characteristics, the longest path from the root to a leaf node is never longer than twice the shortest path from the root to a leaf node – which finally results in $O(\log n)$ complexity. For a detailed description we refer to [5].

However, for the cache an appropriate replacement policy is needed. Mainly, we want to improve cache performance in case of bursts[9] with identical requests where temporal locality is high, e.g. in a simultaneous boot up of many wireless clients. As the operating system internally uses a second cache for frequently accessed blocks, we will replace blocks which seem to be *least recently used by other clients*. Thus, a new referenced block of a request is always placed on the first position of the cache and blocks which have not recently been requested are replaced. An evaluation of the interaction between pre-caching and internal caching of the operation system with possibly improved behavior is deferred for further work.

Side effects

Pre-caching can lead to a synchronization of wireless clients, especially in simultaneous boot-up sequences. Clients which are a few seconds behind could already have cached blocks and then catch up to clients which are still waiting for network

[8] Originally, he called them symmetric binary B-trees; the contemporary name was introduced later.

[9] a temporally accumulated number of requests

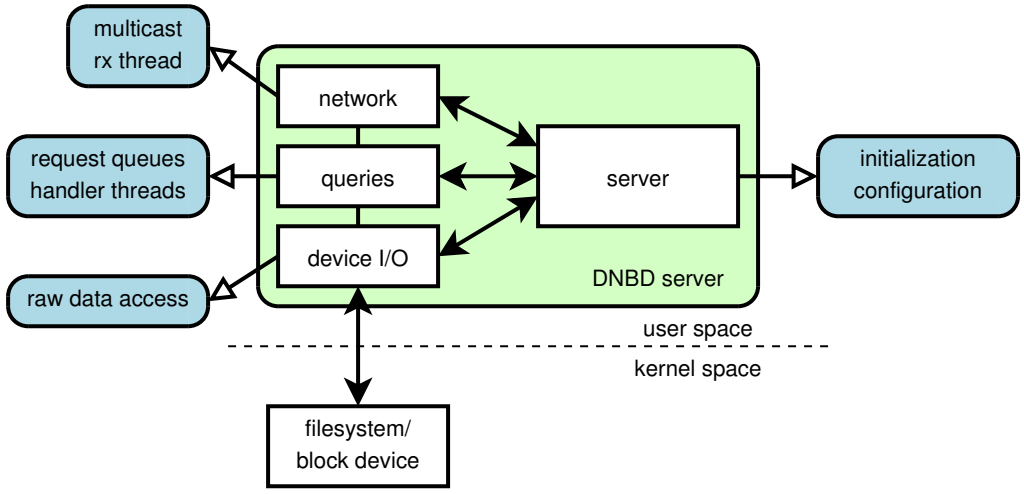


Figure 3.6.: The server is divided into different primary components which are responsible for communication (network), request handling (queries) and block access (device I/O).

I/O. In practice, especially for computers with identical hardware, systems sometimes request the same block in such a short interval (milliseconds) that the server would answer to *each* request and pre-caching can not catch this case. To avoid this behavior, the server uses a circular buffer not only to queue requests but also to check for request bursts.

3.3. Components

DNBD is divided into a server and client part and the client is further split into a kernel module – the actual block device – and a controlling user space application.

3.3.1. Server

The server is a user space application providing a block device[10] to the clients. Whenever a client request is received, it is queued for further processing so that several threads can process requests in parallel. This can increase performance, especially on multi-processor machines, as one thread can retrieve a data block from the exported block device while another thread can send a reply. Internally, the server is separated into different parts according to its tasks (figure 3.6):

[10]The exported data can also be based on a large file instead of a real block device

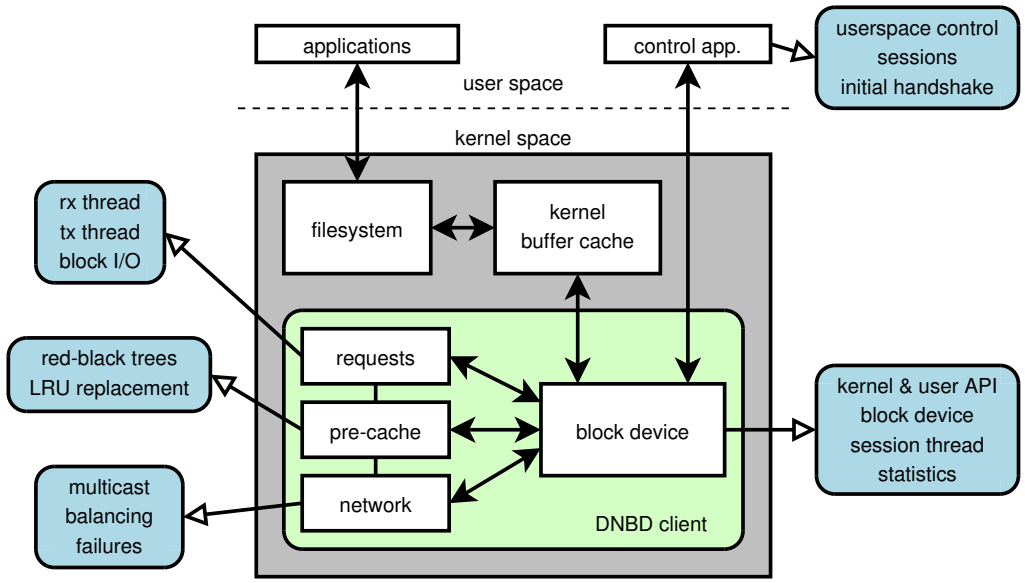


Figure 3.7.: The client part consists of a kernel module and a controlling application. The module takes care of requests, caching and network transmissions. The controlling user space application calls the module to initiate and stop sessions.

- **Device I/O:**
  This part initializes the underlying block device, checks its size and provides functions for seeking to a specific position and reading blocks.

- **Network:**
  The network interface is responsible for setting up the network for multicast and provides a send and receive function.

- **Queries:**
  Requests are queued for further processing. These are processed by a fixed number of threads which analyze the request, read the corresponding block from the block device, generate an answer packet and pass it to the network part.

- **Server:**
  The server part takes parameters and initializes other parts of the application. The configuration is done via command line parameters.

As the the server is multi-threaded, special attention to critical sections are necessary. For example, only one thread may seek *and then* read a specific block at a time; otherwise concurrent block seeks and reads will deliver faulty data.

3.3.2. Client

As mentioned above, the client is split into a kernel module which provides the actual block device and a controlling application. The architecture of the block device driver module is similar to the server, however, it has to perform many more tasks (figure 3.7):

- **Control application:**
  The controlling application is used to initiate a session: A handshake process asks for available servers in a multicast net and, afterwards, configuration parameters of the block device are set (e.g. size, socket etc.). If no problems occur, the kernel module is informed to start a session.

- **Network:**
  Here, the network interface is mainly responsible for analyzing the availability and throughput of the servers and can adapt the priority of used servers. Stalled servers are disabled.

- **Pre-cache:**
  Self-balancing binary search trees are used to cache blocks requested by other clients.

- **Requests:**
  Internally, the kernel module uses two queues for requests that should be submitted (`tx_queue`) and a queue for requests with outstanding replies (`rx_queue`).

The device driver module starts three threads with different tasks. The first thread (`tx_loop`) dequeues requests given by the operating system and submits them to the server. Server replies are evaluated by another thread (`rx_loop`). The third thread (`ss_loop`) sends heartbeat requests to the multicast network and recalculates server weights (cp. section 3.2.5). Additionally, a time driven function periodically checks for requests that have not been answered yet (due to packet losses) and forces a retransmit after a dynamically estimated timeout.

tx_loop

In algorithm 1, we want to take a closer look (in pseudo-code) at the tasks of the `tx_loop` function or thread, respectively. The most important part is the main loop which is executed until it receives the signal to stop from the operating system. However, the thread will sleep until a request arrives. Whenever a request arrives, it will be dequeued from the `tx_queue`. In Linux, a request consists of consecutive blocks to be handled. Therefore, it is checked if blocks at the beginning of a request are already cached and, should that be the case, the request can be

(partially[11]) finished. If the request is not completely finished, remaining blocks will be requested from the server and the request is enqueued as an outstanding request in the `rx_queue`.

Algorithm 1 THREAD `tx_loop`

```
1: THREAD INITIALIZATION
2: // main loop
3: while NO PENDING SIGNAL do
4:    // the thread sleeps until it has to send a request
5:    SLEEP UNTIL A REQUEST ARRIVES
6:    DEQUEUE REQUEST FROM tx_queue
7:    // if requested blocks are already cached, there is no need to send a request
8:    while A BLOCK IS ALREADY CACHED do
9:       COPY BLOCK
10:      FINISH REQUEST PARTIALLY
11:   end while
12:   if REQUEST IS COMPLETE then
13:      FINISH REQUEST
14:   else
15:      ENQUEUE IT AS OUTSTANDING REQUEST IN rx_queue
16:      SEND REQUEST
17:   end if
18: end while
19: // stop thread
```

rx_loop

The `rx_loop` function is the complement to `tx_loop` and takes care of received packets. We use the word *packet* because, up to now, it is not clear, that it belongs to the DNBD protocol. After we verify that the packet is valid and comes from a server (and not from a client) we can update round-trip times of the corresponding server. If the packet is a reply to a read request, further processing is done. If there is no matching request in our `rx_queue`, then cache the transferred block. Otherwise, dequeue the request from the `rx_queue` and copy the data from the network packet to memory where it is expected. Finally, if the request is not completely processed it is requeued in the `tx_queue`, whereupon outstanding requests are transmitted again.

[11] Linux allows the block driver to finish parts of a request. The corresponding blocks have to be contiguous and start with the first one of a request.

Algorithm 2 THREAD `rx_loop`

```
1: THREAD INITIALIZATION
2: // main loop
3: while NO PENDING SIGNAL do
4:    // the thread sleeps until a network packet arrives
5:    WAIT FOR A NETWORK PACKET
6:    // packet integrity and validity is verified
7:    if PACKET IS FAULTY OR WAS NOT SENT BY A SERVER then
8:       CONTINUE AT THE BEGINNING
9:    end if
10:   // we got a valid packet from a server
11:   UPDATE ROUND-TRIP TIMES OF THE SERVER
12:   if REPLY IS ANSWER TO READ REQUEST then
13:      // the packet was an answer to a read request sent by any client
14:      if REPLY CORRESPONDS TO AN OUTSTANDING REQUEST then
15:         DEQUEUE REQUEST FROM rx_queue
16:         COPY BLOCKS
17:         // A request can consist of several blocks. If the amount of data does not
            suffice, the request has to be sent again for subsequent blocks.
18:         if REQUEST IS COMPLETE then
19:            FINISH REQUEST
20:         else
21:            FINISH REQUEST PARTIALLY
22:            REQUEUE REQUEST IN tx_queue FOR SUBSEQUENT BLOCKS.
23:         end if
24:      else
25:         CACHE BLOCKS
26:      end if
27:   end if
28: end while
29: // stop thread
```

4. Implementation

This chapter covers implementational aspects of the Distributed Network Block Device. As most of its source code integrates into the operating system, we explain those points of the Linux kernel interface, which are crucial for an implementation in section 4.1. Main concepts of the user space programs for server and client are outlined in section 4.2. Finally, this chapter concludes with difficulties during the implementation and limitations of this server-client application.

The Block Device has been developed under *Gentoo Linux*[1] – but is not limited to this Linux distribution. Several instances of *User-mode Linux*[2] (UML) have been used, mainly for functional testing, virtual networking and debugging. SuSE Linux 10.0 has been employed for diskless operation.

4.1. Linux Kernel API

The actual Linux kernel 2.6 consists of over four millions source lines of code. Many volunteer programmers have contributed to the development of this open-source operating system, but have not spent much time documenting their work, which can make programming the kernel itself or modules for the kernel quite difficult. Only few books cover implementation details of the Linux kernel, however, a very comprehensive book is [14] which deals with both the relevant concepts and an overview of used data structures and algorithms. A more practical reference for device driver development with many examples is [4].

As Linux in its present form owes its success to the distribution and development over the Internet, this is usually another place to search for further information.

Finally, the most up-to-date source for studying its concepts and algorithms is the code itself. Its enormous size gives the impression that the kernel is of great complexity, however knowledge of its deliberate structure and organization make in-depth studies of its concepts and algorithms possible. Additionally, *hypertext*

[1] `http://www.gentoo.org/`

[2] User-mode Linux is a virtual machine giving the opportunity to run Linux in user space (`http://user-mode-linux.sourceforge.net/`)

documents of the kernel source code with cross references of used identifiers help to analyze dependencies of kernel functions and structures.[3]

4.1.1. Modules

The concept of modules, which allows adding extensions to the running kernel, has been introduced to Linux in version 1.2 and is also used for DNBD's client. Network protocols, file systems, device drivers (e.g. block devices) and many other parts can dynamically be loaded and removed to add more functionality to the system. Without this feature, Linux with a monolithic kernel would have a major disadvantage compared to microkernel systems. Modules can be seen as common programs, but with some specific characteristics. They run in kernel space and thus do not profit of security mechanisms present in user space, e.g. memory protection. However, modules can also be compiled statically into the kernel. The file system drivers needed at boot time, for example, have to be statically compiled into the kernel, as all modules are only available as files in a file system, which can only be read if the driver is already present (chicken-egg paradox).

With modules, Linux as an open-source operating system also provides a way to integrate proprietary drivers. As many companies do not want to release their drivers as source code, they can at least publish binary drivers for their products. Although this concept works in principle, there are major problems – not only of a technical (e.g. fast changing interfaces), but also of an ideological and legal nature.

Usually for the development phase device drivers are intended to be external modules and minor changes of the source code do not implicate a complete recompilation of the kernel as it would be in the statical approach. For flexibility, the DNBD kernel driver also compiles as an external module which can dynamically be loaded when necessary. A short introduction to device driver programming for modules is given in the next sections.

Module Representation

For module development the kernel offers an interface defined in `<linux/module.h>`. A module has to implement at least two functions, one for its initialization and one for its removal. The special kernel macros `module_init` and `module_exit` are used to specify these functions. The macro `MODULE_LICENSE` is used to declare the license of a module. Modules integrated in the kernel sources are usually copyrighted with the *GNU General Public License* (GPL).

[3] `http://lxr.linux.no/`

Another important header file is *<linux/init.h>* which defines the `__init` and `__exit` tokens. These tokens are hints for the kernel and compiler how to handle the functions. For example, the first one shows the kernel that the function is only needed for initialization and can be dropped afterwards; with the second one the compiler knows that this function can be dropped, if a module is compiled statically in the kernel and unloading is not possible.

The following code is a trivial example of a module:

```
#include <linux/init.h>
#include <linux/module.h>

static init __init init_function(void)
{
  /* Code for initialization */
}

static void __exit exit_function(void)
{
  /* Code for cleaning up */
}

module_init(init_function);
module_exit(exit_function);

MODULE_LICENSE("GPL");
```

Most of DNBD's functionality is implemented in a kernel module. Here, the initialization function has far more to do. The DNBD driver sets up `proc` entries giving the opportunity to query state and statistical information for a device, e.g. `/proc/driver/dnbd/dnbd0`. Afterwards, data structures for the device, cache and server management are filled with default values and a fixed number of block devices (section 4.1.3) are registered with the kernel. Conversely, the `exit` function unregisters the device and `proc` entries and frees reserved memory, as the case may be.

4.1.2. Device Drivers

An important part of the kernel are device drivers which make it possible for the operating system to communicate and interact with hardware devices. As every piece of hardware has specific characteristics, device drivers act as an abstraction layer between well-defined interfaces of the operating system and non-generic methods of the hardware. Figure 4.1 shows the integration of hardware in the

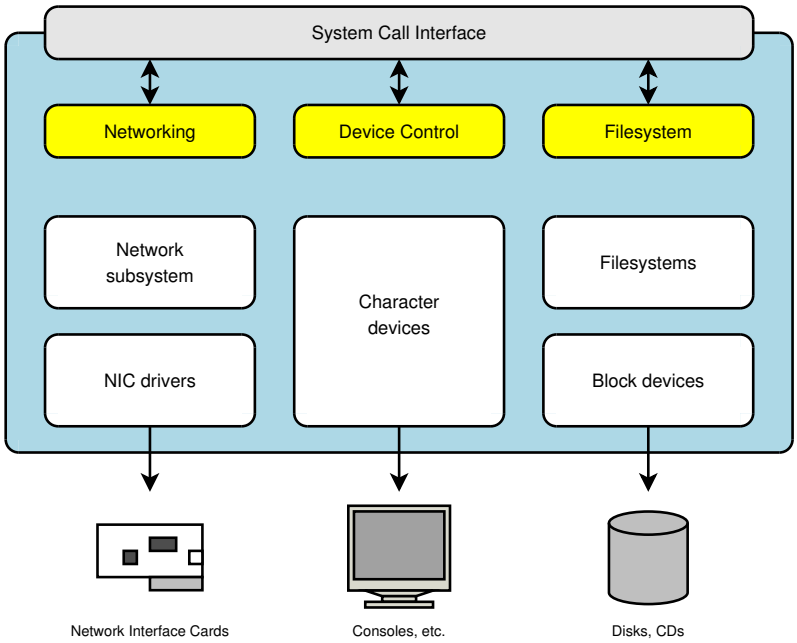


Figure 4.1.: There are three fundamental device types in Linux: network, block and character devices. They provide an abstraction layer for hardware and a generic interface for the operating system. Implemented as a module they can be dynamically added and removed at run-time.

operating system and the different layers of abstraction before the interface is exposed to user space and thus allows indirect access to the hardware through system calls.

Device driver development is usually considered a challenge for several reasons. In contrast to application development, the programmer requires in-depth knowledge of the platform and operating system, including methods for process and memory organization, interprocess communication, locking etc. In addition, the hardware specification of the device should be known, to avoid a laborious reverse engineering process. Device drivers have to be implemented very accurately, because bugs can freeze the whole system and debugging of drivers usually requires an adaption of the operating system.

DNBD is not an abstraction of a hardware device, nevertheless, it has to consider the aspects mentioned above. Device drivers perform their tasks transparently to the user, but Linux can also offer indirect access to devices with help of device files.

Device Files

Linux uses special files to access hardware devices. These files are not directly connected to the hardware, but linked to the corresponding device driver. In a

common Linux system, these special files can be found in the `/dev` directory. Some typical entries are `sda` (SCSI disk), `tts/0` (First UART serial port) and `zero` (Null byte source):

```
# ls -la /dev/sda /dev/zero /dev/tts/0
brw-rw----  1 root disk 8,  0 Nov 13 16:58 /dev/sda
crw-rw----  1 root tty  4, 64 Nov 13 16:58 /dev/tts/0
crw-rw-rw-  1 root root 1,  5 Sep 11 12:44 /dev/zero
```

The main differences to regular files is the first character in each row which shows the type of the device (block or character device); and each device possesses a major and minor number as unique identification (e.g. 8 and 0 in the first row). Thus, files of the same device type (e.g. partitions of a hard disk) usually all have the same major number, whereas the minor number is incremented.

The representation of device numbers in the kernel is defined in `<linux/types.h>`. For this a 32-bit `dev_t` type is split into a 12 bit set for the major number and the remaining bits for the minor number. This variable should be accessed by macros given in `<linux/kdev_t.h>`: `MAJOR(dev_t dev)` and `MINOR(dev_t dev)` for the major and minor number respectively; and `MKDEV(int major, int minor)` should be used to turn them back into a `dev_t` type.

The kernel also offers various routines for allocating and freeing device numbers, both statically and dynamically. The latter is convenient for new drivers, because it will not interfere with existing numbers already reserved for other drivers. An overview of currently used major and minor numbers can be found in the kernel sources (`Documentation/devices.txt`). DNBD uses the dynamic approach and leaves finding a free major number to the kernel. This results to entries like:

```
# ls -la /dev/dnbd*
brw-r-----  1 root disk 253,  0 Dec  5 11:27 /dev/dnbd0
brw-r-----  1 root disk 253,  1 Dec  5 11:27 /dev/dnbd1
```

In the next section, we will consider the representation of block devices and their corresponding data structures in the Linux kernel.

4.1.3. Block Devices

We have seen that special files are either for character or block devices.[4] Character devices are not used in this work, but – for better comparison with block devices – they are briefly outlined here. Character devices are usually accessed as a stream

[4]Named pipes are a third type – but they are not used for device drivers.

of bytes. A keyboard driver for example, supplies a key code each time a key is pressed. A serial device (like `/dev/tts/0` mentioned above) can send or receive data from a serial line. In contrast to regular files, it is neither possible[5] nor desired to seek within these special files.

Block devices mainly differ from character devices in two points: Block devices allow arbitrary access to positions and data is always transferred in blocks of a fixed size. Even if only one byte is requested, a complete block has to be transferred to memory. The kernel uses various interrelated structures and algorithms to handle and improve access to block devices. The most important mechanisms are *request queues* and *caches*.

Buffer Cache

Most operating systems use buffers to improve the access to block devices. The principle is simple: Whenever a block has to be read, it is first checked if it is already buffered in memory and access to the corresponding block device can be omitted. If it is not present in a buffer, the block has to be read from the device and is added to the buffers with a LRU (least recently used) replacement policy. Modified blocks are kept in memory and marked as *dirty*; these are written back to disk after a given time – usually together with other modified blocks. Linux utilizes unused memory for dynamic buffer allocation. If the memory is needed by applications, the number of buffers is reduced. It also tries to map buffer pages directly to virtual pages of a process in order to prevent copying such pages [2, Ch. 4.3].

The kernel buffer cache and DNBD's internal cache are separate entities: The buffer cache only stores blocks which *have already been* requested by applications, whereas DNBD caches blocks which *could* be requested by an application.

Request Queues

A request queue is more than a queue of block I/O requests. In fact, the kernel uses complex structures and algorithms for its request queue management. As hard disks (represented as block devices) are usually slow, the kernel tries to make the best of the physical constraints of this data storage. These physical constraints are that the data of files can be scattered over different platters of a hard disk and seeking to these blocks can be seen as one of the most time-consuming operations in a computer.

I/O schedulers are used to achieve optimal performance in this context. They keep track of requests in order to sort and merge them for maximum performance – or

[5]Exceptions are frame grabbers (amongst others) which allow operations like `lseek`.

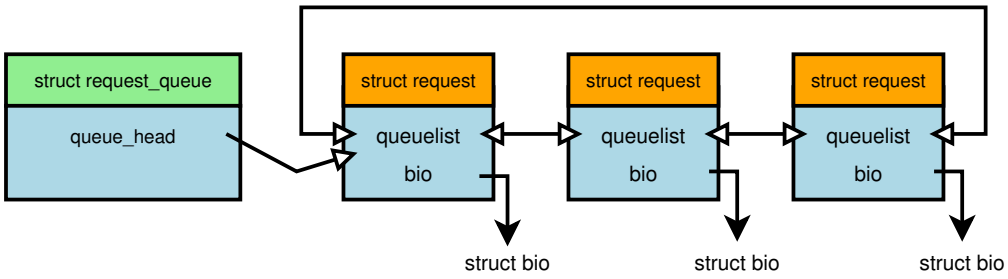


Figure 4.2.: Read and write requests are collected in *request queues*. This structure includes a pointer to a doubly-linked list which contains the requests. Each request has a pointer to a so-called `bio` (block I/O) structure which maps a block to a page instance in memory (figure 4.3).

optimize requests to achieve better response times in an interactive system. Thus, the *anticipatory scheduler* intentionally delays read requests if it seems likely that a process will submit another request soon; the *deadline scheduler* can merge and reorganize requests that are submitted before a certain time interval expires [13]. For some block devices and especially for DNBD, schedulers usually play a minor role, e.g. memory block devices like USB storage have no mechanical restrictions, whereas DNBD would split merged requests to a suitable size for network transfers and also delaying requests has no advantages. However, this refers only to DNBD's client and not the server which could export a mechanical hard disk.

The `request_queue` structure in the Linux kernel consists of over 40 elements (defined in `<linux/blkdev.h>`). It starts with components to keep requests in a doubly-linked list and also pointers to functions which can, for example, create, resort and merge requests. Further parts are unplug elements to realize timer mechanisms used to unblock queues after a given time; this situation of blocked request queues can occur, if the system is overloaded. The last important components are hardware related values, such as the physical sector size (which is usually 512 bytes).

Requests

Finally, the requests themselves are kept in a separate structure and the request queue keeps a pointer to a list of requests as mentioned above. The most important components of this structure are the consecutive sectors to be handled, information where blocks are transferred from and to memory (BIO or block I/O) and various variables or flags concerning the I/O scheduler. The kernel can join requests of adjacent sectors for either reading *or* writing, but not a mixture of both.

The so-called `bio` structure of a request includes a set of segments which is used to either transfer blocks from a block device to an in-memory buffer or vice versa;

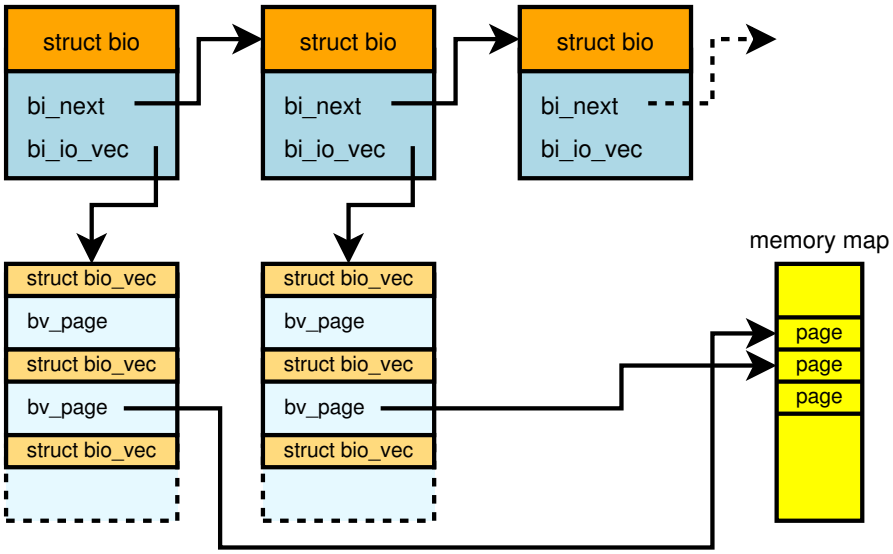


Figure 4.3.: A request holds a pointer to a list of `bio` structures, whereas each BIO has a pointer to a vector array with the corresponding memory page information. The kernel uses these structures to transfer data blocks from a block device to memory or vice versa.

or more precisely, BIOs are organized in lists, each with a pointer to a vector array `bi_io_vec` (including its size) which contains a pointer to the corresponding `page` instance.

For a better understanding, the correlations between these structures are outlined in figure 4.3. For implementations, the kernel provides some macros to access and handle these structures more easily, including functions for page and memory mappings, loops over the segments, etc.

DNBD uses these specific `bv_page` pointers to transfer blocks. The size in bytes to be requested is given by a `bv_len` variable which is related to the block size[6] of the device. When using a file system on top of DNBD, this relationship should be considered, as file systems can presume a minimum block size. Then the corresponding replies of the server, whose packet sizes are greater than the maximum transfer unit of the network, have to be fragmented by the IP protocol. This behavior is usually undesirable[7], but fortunately, most file systems allow manually setting the minimum block size at file system creation. Incidentally, fragmentation also appears with TCP and not only for UDP. For further work, it is planned to allow splitting of such blocks to arbitrary sizes and independent from the file system. However, in this case the device driver has to keep track of the parts of each block that are already transferred, before it can signal completion to the operating system.

[6] Allowed block sizes for Linux are 512, 1024, 2048 and 4096 bytes. Some devices have even a lower bound, e.g. 2 kB.

[7] If one of the fragments of a packet gets lost, all fragments have to be retransmitted.

4.1.4. Networking

Linux has had early support for various network protocols, including the Internet protocol (IP), and the network implementation covers a major part of the kernel source code. Its network model is very versatile and provides many extensions for special applications, e.g. QoS[8], network emulation, cryptography, routing protocols for ad-hoc networks and much more.

Sockets

The main interface for network communication are sockets. With the help of sockets, the system provides methods for opening and closing a communication channel and for data exchange which can be used for interprocess communication or network communication. Sockets are not limited to the Internet protocol (IP) – they support all different kinds of communication devices (e.g. bluetooth) and protocol families (e.g. AppleTalk, IPX etc.). In DNBD, socket setup is done in user space for both the server application and client part. The controlling application for the kernel module `dnbd_client` is responsible for creating a socket and setting network parameters defined in `<sys/types.h>` and `<sys/socket.h>`.

Multicast

As DNBD uses multicast communication, it is also necessary to adjust some attributes of the socket. The first step is to tell the system that we want to join a multicast domain. Then, the kernel will send special IGMP[9] messages to inform multicast routers that a member has joined or left a multicast group. If multicast is needed in different networks the *time-to-live* (TTL) value has to be set to the number of routers (or hops) which have to be passed. A TTL of zero will prevent packets from crossing subnet borders. Finally, it should be avoided that packets which are sent to a multicast group are received by the sender itself. This behavior is called multicast looping and may be necessary for some applications,[10] but here it is suppressed. DNBD changes the default value of the TTL for multicasts to 64, as usually multicast routers decide themselves – according to a specific configuration, if they forward a packet or not. However, the default value of 0 in Linux would result in not forwarding this packet, at all.

[8] *Quality of Service* tries to provide better service (bandwidth, delay, etc.) for specific network services.

[9] The *Internet Group Management Protocol* (IGMP) is used to organize multicast groups

[10] In this way, Internet telephony applications can be tested without need of a remote station (echo test).

4.1.5. Threads, Synchronization and Locking

As a multi-tasking operating system, Linux is able to run multiple multiple processes or threads in parallel. For this the kernel defines in `<asm/processor.h>` the function `kernel_thread` in order to start kernel threads. Its three parameters consist of the pointer to the threaded function, a pointer to a data structure which is given as argument to this function and finally some flags to inform the kernel which characteristics of the calling process should be inherited. Thus, for DNBD, the thread for sending requests (function `tx_loop`) is started by

```
kernel_thread(tx_loop, dnbd, CLONE_FS | CLONE_FILES);
```

where `dnbd` is the structure containing all important information about the corresponding block device and `CLONE_FS | CLONE_FILES` inform the kernel that caller and child process share the same file system (`FS`) information and file descriptor table (`FILES`).

Altogether, DNBD starts three kernel threads for sending requests (`dnbd_tx_loop`) and receiving their answers (`dnbd_rx_loop`) and – for statistical calculations – a periodical session thread (`dnbd_ss_loop`). However, it is important that threads synchronize their work – or in other words – sleep when there is nothing to do and give execution time to another thread. This can simply be done by calling the scheduler or waking up a waiting thread. Thus, the sending thread is sleeping until the kernel requests the device driver to transfer some blocks. Similarly, the receiving thread continues its work whenever a packet arrives over the network.

The kernel offers functions and macros (in `linux/wait.h` and `linux/sched.h`) to implement synchronization by sleep and wake up calls. Before a process goes to a sleep state, it has to place itself in a so-called wait queue – a common way to make it possible to wake it up again from another thread.

Threads are usually not interrupted, unless they voluntarily release their execution in favor of another thread (e.g. by the scheduler) or either a software or a hardware interrupt occurs. In DNBD, a time dependent function `dnbd_rexmit` is executed to resend requests after a specific timeout. As the kernel timer is interrupt driven, it can occur that both the retransmit function and the sending thread try to concurrently access the same data structures and request queues, respectively. Here, the kernel provides several methods to avoid concurrent access to a critical section with so-called semaphores and spinlocks. The main difference is that semaphores can sleep until the lock is released, whereas spinlocks keep a processor waiting or *spinning*. For single processor systems, spinlocks are usually defined as empty operations – unless the kernel supports preemption, where the operating system can stop or preempt a currently scheduled task in favor of a higher priority one. In this case, activating a spinlock is generally the same as temporarily disabling preemption. Although the principle of locking is simple, misuse (e.g. sleeping while holding

a spinlock) can lead to laborious debugging of code, as the operating system can reach an undefined state or even freeze.

4.2. User Space Programming

Until now, we have only considered implementational aspects of the client part of DNBD – or more precisely, the block device driver. However, this module would be useless without its controlling user space application and, of course, a server which answers its requests. Linux system programming is covered in [11] which provides concepts and functions used for DNBD's user space applications.

4.2.1. Client Control

The controlling application for the client has two simple functions: It takes configuration parameters given by the user (multicast net and block device), initiates a handshake procedure and finally, after setting configuration parameters, it calls on the network block device to start a session (cp. section 3.3.2).

The communication between client module and application is done via so-called `ioctl` calls. The `ioctl` function is a system call which manipulates the underlying device parameters of a special file outside the usual read and write of data. Its arguments are a file descriptor, a device-dependent request code and a pointer to memory. Several codes (integer macros) for DNBD have been defined for internal communication, like `DNBD_SET_CAPACITY` (set device capacity), `DNBD_DO_IT` (start a new session) and `DNBD_DO_DISCONNECT` (end a session).

4.2.2. Server Application

The server is a self-contained application and also takes command line arguments to set up the multicast network, the underlying file (or device) and a unique server identifier (cp. section 3.3.1).

Depending on the request, the server either answers with configuration parameters (during the handshake procedure) or delivers a data block. The server is designed to be multi-threaded, i.e. requests are queued and then processed by multiple threads, which can give performance benefits on multi-processor systems. However, queuing and context switching result in additional overhead and, in practice, it depends on the underlying hardware if performance is increased.

4.3. Difficulties

In user space applications the operating system usually takes care of memory protection, access to devices and can even *kill*[11] stalled applications. In kernel space, there are only few safety mechanisms – and wrong code can have a massive impact on stability and the integrity of the system. The system can even freeze and then it is hard to figure out what exactly caused this behavior.

As mentioned at the beginning of this chapter, *User-Mode Linux* (UML) was used to test the implementation. This virtual machine environment makes it possible to debug even the operating system and allows fine-grained analysis, monitoring and manipulation of code [8]; however, the UML kernel works slightly different to a common kernel and many hardware devices, e.g. network adapters, are virtualized. Although the UML architecture is part of the main kernel, it was impossible to compile it with specific configurations of a kernel version. The result is that a problem which occurred in a real environment could not always be reproduced under UML.

In this case, the most simple way of tracing and monitoring a program is the use of the `printk` function which can be placed, for example, at the the beginning or end of an function to show the content or pointers of input and output parameters. The `printk` function takes as its parameters a so-called *log level*[12] and string arguments for a message. The `printk` is widely used in the kernel in order to inform the user of specific hardware information and errors like the presence of a new device (e.g. a USB stick) or even I/O errors when accessing a hard disk. The *debugging by printing* method is often laborious because manual changes of the source code are necessary.

During the implementation process of this work, some time consuming fault diagnostic processes were not related to DNBD itself, but to other parts of the kernel. When *squashfs*[13] was used on top of DNBD, block transfers suddenly stalled. Although this problem seemed to be DNBD related, it turned out to be the faulty usage of a semaphore in squashfs that caused this behavior. Fortunately, a patch for this bug could be found on the squashfs project page. Another problem occurred with the CFQ I/O scheduler in form of a freeze of the operating system kernel. It could only be reproduced with the original kernel of SuSE 10.0 and was supposably related to a bug with *elevator switching and corrupt request ordering during a flush sequence* in CFQ.

[11] A process can be forced to terminate by sending the *kill* signal to the operating system.
[12] a hint of the severity of the problem
[13] Squashfs is a compressed read-only file system (`http://squashfs.sourceforge.net/`).

4.4. Limitations

DNBD can be utilized in both wireless and wired networks. State-of-the-art network adapters offer raw data rates of 1 Gbit/s or even more. Depending on the data rates of the network interface, CPU speed, but also the I/O scheduler, file system and configuration parameters (e.g. read ahead) of the block device, packet bursts can occur, which can congest the internal kernel buffers for received and sent datagrams. In practice, dropped packets in case of full buffers are unlikely, as the operating can process packets much faster than wireless links can deliver them. However, in order to avoid this drawback a distributed algorithm would be necessary, which coordinates requests and the implied replies so that no congestion would occur on either the servers or clients.

Another limitation of DNBD is IPv6 communication. The Linux IPv6 implementation is sophisticated and user space applications can revert to it with the common network or socket functions (appendix C). In contrast to IPv4, multicast addresses include a field for the area of validity, e.g. for node, site, organization or even global wide multicasts. However, DNBD requires some minor adaption and testing which could be done during further development.

Finally, security mechanisms are not included in DNBD's request-reply protocol. It is questionable whether this is really necessary in this middleware protocol, as authentication and encryption can also be included on other layers, e.g. *virtual private networks* on network layer or user authentication with *LDAP*[14]. However, the problem of secure key exchange remains, due to the risk of a man-in-the-middle attack. This is a general problem – not only for DNBD or diskless clients. If security plays a decisive role for untrusted environments, one should revert to keeping keys, passwords, certificates, etc. on other media like smart cards or flash drives.

[14]Lightweight Directory Access Protocol

5. Experiments

This chapter deals with experimental results. We compare our DNBD implementation to other network storage solutions. The benchmarks were accomplished with standard benchmark and monitoring tools available under Linux. For some experiments we used quality of service capabilities of the kernel and simulated data loss and delay with a network emulator (appendix C). As a proof-of-concept for scalability and to gather experimental results, a setup of diskless computers has been used.

5.1. Basic Comparisons

5.1.1. Network traffic

A SuSE 10.0 System was adapted for operation with a network storage with help of the *Linux Diskless Clients* project (appendix C). Its initial ramdisk evaluates various kernel boot parameters, and one and the same image can be used for different boot methods, e.g. DNBD, NFS, NBD and others. Figure 5.1 shows the amount of traffic (incoming and outgoing) on the server for a *single* client.

The first measurements with NBD were surprising (third bar). Although its TCP protocol entails a little more overhead than UDP which is used by DNBD, the traffic generated was much greater than expected in relation to DNBD. The reason for this overhead is the *read ahead* (RA) behavior of the kernel. In order to increase performance of a block device, the kernel not only reads a block at a specific position, but also reads some subsequent blocks. This is useful mainly for large files which are mostly written in successive sectors of the underlying block device. On average, it can be better to read some blocks in advance than only request necessary blocks and, later, after the read head of a hard disk has moved (due to requests of another process), seek to its former position. To minimize traffic, DNBD disables read ahead per default, but for NBD it has to be manually disabled (second bar).[1]

DNBD entails less traffic compared to NBD due to the slightly lower overhead of its UDP protocol. The different sizes of the TCP and UDP headers in relation to the

[1] e.g. with the command: `hdparm -a 0 /dev/nbd0`

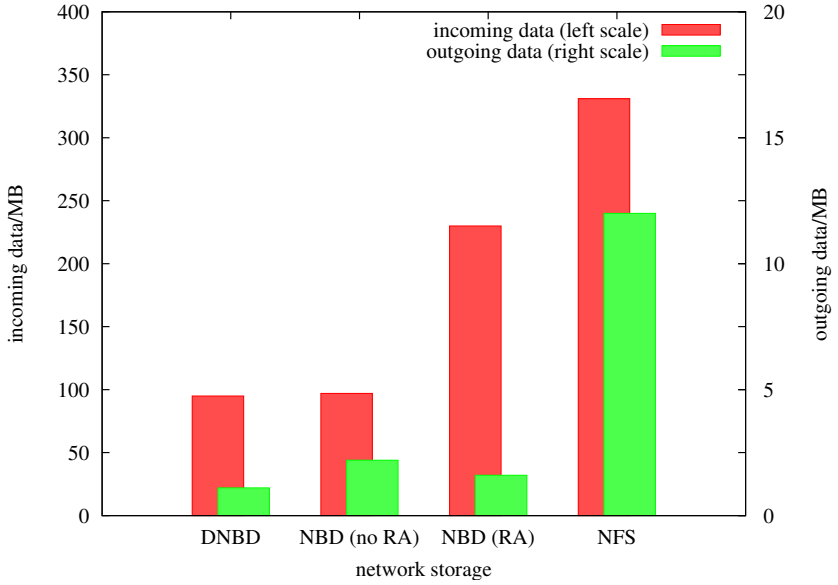


Figure 5.1.: The figure shows data amounts for diskless boot with DNBD, NBD and NFS. Read-ahead (RA) has to be manually disabled for NBD to lower unnecessary data transmission.

small size of a block request appear in the amount of outgoing traffic with NBD and DNBD. However, both for NBD and DNBD outgoing traffic is almost negligible, as it is approximately 1% of the incoming amount of data. For comparison with a network file system, NFS with default values lies far behind the network block devices in this experiment. Some configuration values could be tweaked, but it is very doubtful that the amount of traffic can be lowered decisively: NFS is a network file system including methods for concurrent file access for different clients and file operations are piggy-backed on RPC calls which are more complex than the headers of the simple DNBD protocol.

Network traffic can decrease even more, if DNBD is equipped with a compressed file system. *Squashfs*, as we mentioned in the chapter before, can package and compress a file system to a single file – which is exported at the DNBD server. This approach is better than on-the-fly compression of data packets, which would give additional overhead to the server.

5.1.2. Throughput

In this section we evaluate throughput measurements of our block device and use the same environment as in section 5.1.1, except that bandwidth is limited to 8 Mbit/s, which should be similar to the effective rates of an IEEE 802.11b access point. Data exchange is done between one server and a single client. In order to get comparable results and to minimize traffic, read ahead was disabled, the internal

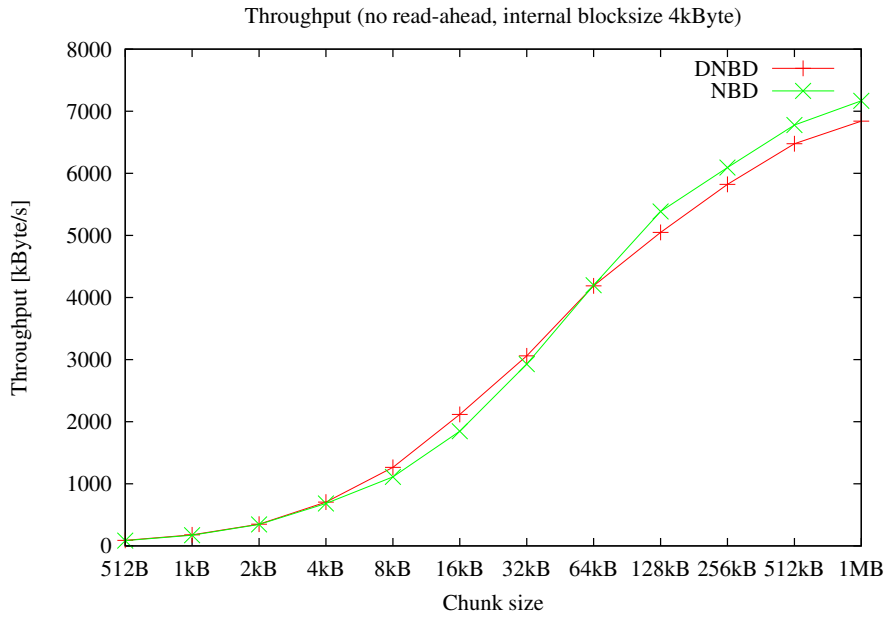


Figure 5.2.: Throughput with reads of different chunk sizes at random positions is almost equal for both DNBD and NBD.

block size was set to 4 kByte for both DNBD and NBD and only a single thread is processing requests for DNBD. In figure 5.2 data chunks of different sizes have been randomly read from the server.

For chunk sizes smaller than 64 kByte, DNBD performs slightly better; however, NBD shows better performance when more linear data in a chunk is requested. This could originate from a more complex implementation of the server, which has to include locking mechanisms for parallel thread processing.

5.2. Pre-caching Evaluation

As a proof-of-concept, we have performed standard tests which affect caching of requests. When many clients access blocks randomly on the underlying block device, the cache hit rate converges against $r_{\text{cache hits}} = \frac{\text{cache size}}{\text{block device size}}$ (cache size $\leq$ block device size), as expected.

As a practical example, we test scalability of DNBD in a wireless-like environment and the maximum output bandwidth of the server is again decreased to 8 Mbit/s. However, our attention is turned to data amounts and boot-up time of those clients, which are most important in practice.

The test system of a single workstation has a size of almost 100 MByte for boot-up with DNBD (figure 5.1). Cache size is generously set to 512 MByte and the

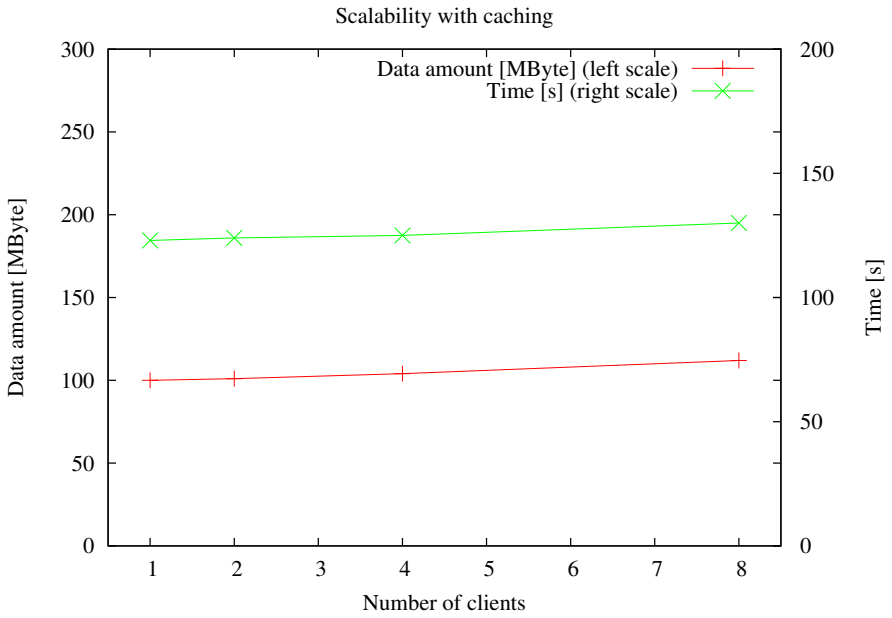


Figure 5.3.: Scalability with pre-caching has been measured for a small computer pool with up to eight workstations. Both data amounts and boot-up give only small additional costs for (few) further clients. TCP solutions would give $O(n)$ complexity with n clients.

corresponding file is kept in main memory (1 GByte) on the client. As presented in figure 5.3 simultaneous boot-up of up to eight clients entails only slightly greater traffic and boot-up time. How far this roughly linear dependency is valid for further clients, in practice, could not be evaluated in this work, as large-scale experiments would be necessary.

5.3. Destination Designation

For destination designation, servers are selected according to their weights, as we explain in section 3.2.5. We have chosen a value of $\beta = 99.5\%$ which gives a good tradeoff between the average round-trip time and new measurements. However, we have observed that overall performance can noticeably decrease, if server characteristics concerning delay and loss differ on a large scale, e.g. one server has a delay of 1 ms and a second one has 30 ms. Indeed, the probability is small that the slow server will be selected, but clients have to wait longer for a reply, if this is the case. Under real conditions, such an imbalance should not occur, however, in this case it could be advantageous only to use the server with the best weight and set *load-balancing* aside.[2] Another proposal, which could be evaluated in future work,

[2]This destination designation method is already implemented and can be uncommented for compilation. The possibility to select this behavior from user space will be given in a future release.

is to give a lower average round-trip time of (fast) servers more relevance with an appropriate value for l, e.g.

$$W_i = \frac{1/SRTT_i(n)^l}{\sum_{j=1}^{S} 1/SRTT_j(n)^l} \qquad (i = 1, \ldots, n; l \in \mathbb{R}, l \geq 1)$$

and disable servers which fall below a certain weight.

5.4. Bad Links

Wireless networks usually entail delayed, intermittent and unreliable communication. We will evaluate DNBD's flow control mechanisms in simulated environments. Netem (Network Emulator) offers the possibility to establish network interfaces with packet loss and delay. This has been done for both experiments on the server; thus, a loss of 1% simply means that there will be no reply to a request with a probability of 1% – and not that a packet disappears with an equal probability of 1% on both its way from the client to the server *and* from the server to the client.

5.4.1. Packet loss

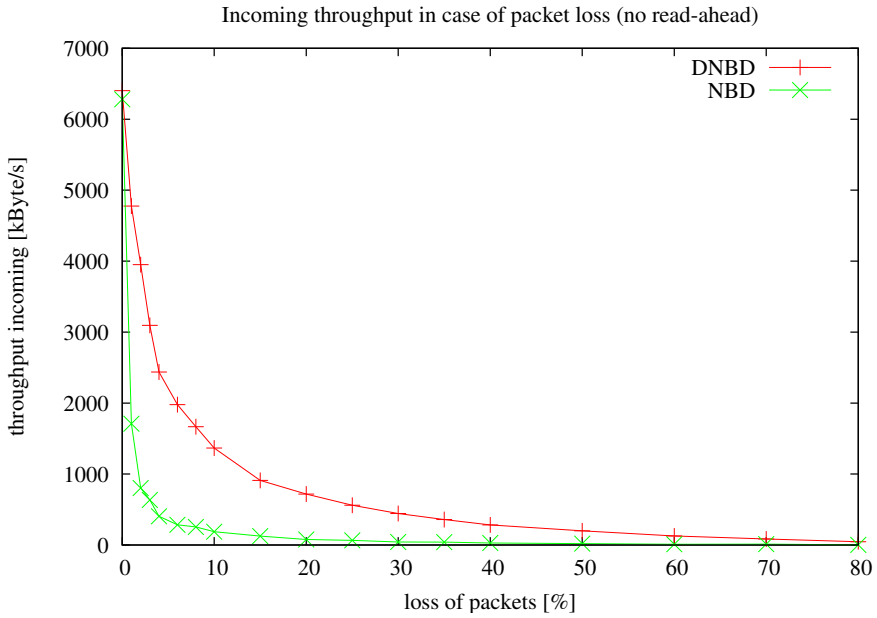


Figure 5.4.: Packet loss has drastic impact on communication performance. Shorter timeouts for retransmits could improve throughput, would however entail more traffic on the network.

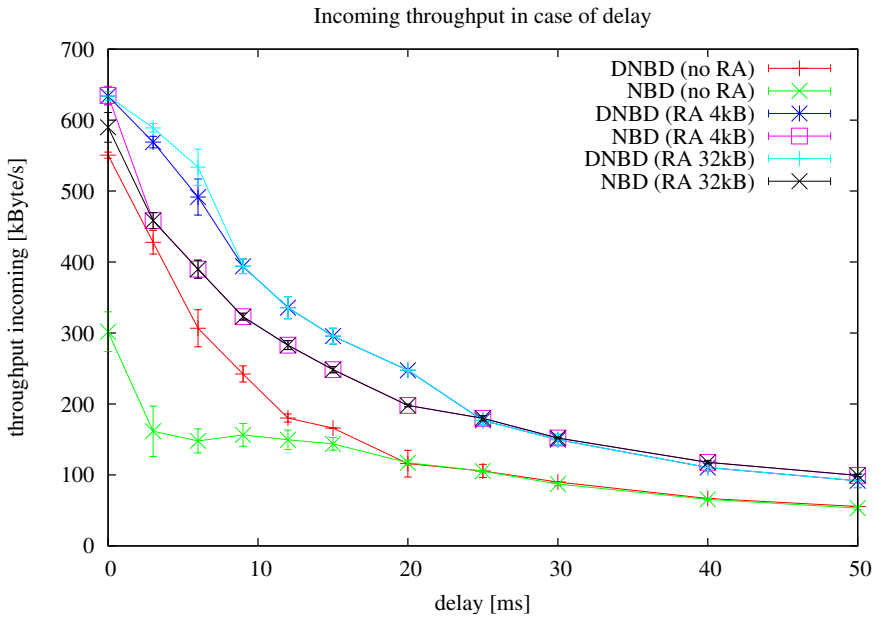


Figure 5.5.: This figure shows throughput with several read-ahead values in dependence of different packet delays. Small delays give more scattered results which could be due to the network emulator.

Figure 5.4 shows linear reading performance (without read-ahead) of the block device measured depending on different packet loss probabilities. Here, the rapid decline of both DNBD and NBD performance at little packet loss is noticeable. With less conservative timings (cp. section 3.2.3), throughput would become better but would entail more overhead for the network.

5.4.2. Packet delay

Throughput measurements with linear reads in dependence on different packet delays are shown in figure 5.5. Different read-ahead values are used, although this would entail additional overhead for non-linear reads. A peculiar behavior is noticeable for delays up to 20 ms. We assume that the network emulator is responsible for that, as short delays make higher demands on software timers.

5.5. Summary of Results

DNBD is a complex system and there are many external factors that influence experimental results. Regarding the kernel, there are internal buffer caches, the I/O scheduler, read-ahead of blocks which should be disabled to get reasonable

and unadulterated results. However, changing read-ahead, for example, can have a major impact on the performance of reads.

As usual, unreliable transmission paths entail an overhead for every protocol and TCP was originally not developed for wireless communication. However, for reliable networks, different implementations offer similar performance. In our experiments, DNBD's flow control does better than TCP for lossy and delayed networks. The results concerning scalability are very promising, but further large-scale experiments have to be done.

6. Concluding Remarks

In this thesis we presented a distributed network block device (DNBD) which minimizes network traffic for wireless clients. At first, we concentrated on the design of such a network data storage considering unreliable communication. A combination of several approaches, e.g. one-to-many semantics in communication, pre-caching, flow control mechanisms and server replication, were considered to improve robustness, scalability and performance in radio networks. The results were used to present the details of the architectural components, consisting of a server and a client module including a control application.

The second part tackled important implementational aspects of our network block device. As the DNBD client module becomes part of the operating system, we considered interfaces and data structures provided by the Linux kernel and outlined various used concepts of system programming, like network sockets, threads and locking mechanisms.

Finally, experimental results of DNBD were evaluated in different environments including characteristics of unreliable radio networks. Concerning performance, we attained higher data rates in lossy networks with our own flow control in UDP compared to NBD, which uses the TCP protocol. We tested scalability for small diskless pools and the results were convincing.

DNBD is not limited to wireless clients and could easily be used to establish a multimedia library, for example. Many applications use multicast communication to stream video or audio content which is played simultaneously by clients. As DNBD has an integrated cache, multimedia contents can be displayed despite a certain delay (depending on cache size) without starting a new communication stream.

Therefore, and as another practical suggestion, DNBD could be used as middleware for a Linux PDA which acts as interactive guide for visitors to a museum. The wireless embedded device could simultaneously play media contents about sculptures and other objects for a higher number of people.

Further work could include long-term tests under real conditions and adaption to flow control. Additionally, the implementation could be improved to include IPv6 support and more configuration parameters for fine-grained customization.

A. DNBD Usage

Compilation

Compilation of the code is simply done by running `make` in the `dnbd` directory of the unpacked sources.[1] The kernel module is build for the currently running kernel – otherwise the `KDIR` variable of the `./kernel/Makefile` has to be customized. All parts should compile without problems for current kernels with a version number higher than 2.6.13. Based on experience, internal interfaces and data structures of the kernel sometimes change and this can lead to errors when compiling the kernel module.

Server

Usage

The usage of the server is simple. If it is executed without parameters, necessary command line parameters are listed as follows:

```
$ ./server/dnbd-server
dnbd-server, version 0.9.0
Usage: dnbd-server -m <address> -d <device/file> -i <number>
                   [-t <threads>]

description:
  -m|--mcast     <multicast address>
  -d|--device    <block device or file>
  -i|--id        <unique identification number>
  -t|--threads   <number of threads>
```

- `mcast` (multicast address):
  The multicast address is an IPv4 address in the subnet 224.0.0.0/8 (class D). However, most addresses are reserved for special purposes (e.g. 224.0.0.0 through 224.0.0.255) and the range 239.0.0.0/8 should be used for experiments.

[1] The source code can be found on the attached CD-ROM under `./dnbd`.

- `device` (block device or file):
  The argument is either a path to a block device or a regular file, e.g. `/dev/hda6` or `/tmp/cd-image`.
- `id` (unique identification number):
  A unique identifier is assigned to each server. A common way is to give incremental natural numbers starting with 1.
- `threads` (number of threads):
  Number of threads to be started. (default 1)

A typical execution of the server with command line parameters for the server could be

```
$ ./server/dnbd-server -m 239.0.0.1 -d /tmp/image -i 1
```

Client

Kernel module

The kernel module is loaded using the command `insmod ./kernel/dnkd.ko`. It registers itself with the kernel as block device and some new (logically independent) device files should appear (`/dev/dnbd1`, `/dev/dnbd2`, ...).

Usage

Just as the server, the client shows possible configuration parameters when started without arguments.

```
$ ./server/dnbd-client
dnbd-client, version 0.9.0
Usage: dnbd-client -d <device> -b <address> [-c <file>]
    or dnbd-client -d <device> -u
    or dnbd-client -d <device> [-c <file>]

description [binding]:
  -d|--device    <device>
  -b|--bind      <multicast address>
  -u|--unbind
  -c|--cache     <file>
```

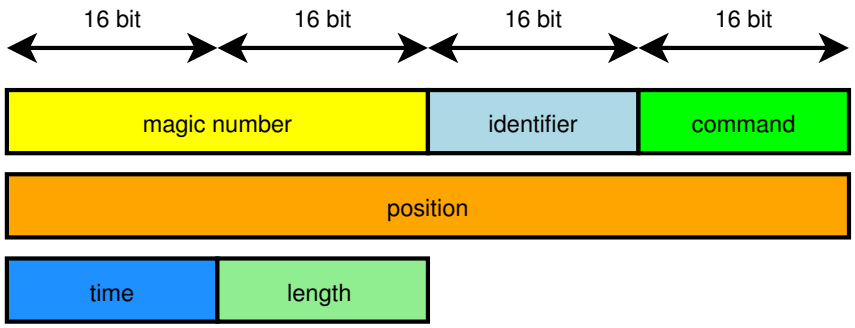
- `bind` (multicast address):
  The DNBD device is bound to the given multicast address which corresponds to the used server address.
- `unbind`:
  This option frees or unbinds a device; this must be done before unloading.
- `device` (block device):
  The argument gives the DNBD block devices to be used for binding, e.g. `/dev/dnbd1`.
- `cache` (cache file):
  An (empty) file has to be created in order to use it as a cache file, e.g. `dd if=/dev/zero of=cache bs=1M count=64` creates a file with a size of 64 MB. The block size of the cache corresponds to the block size of the DNBD device. A cache file should be set after mounting the file system on top of DNBD as mounting changes the block size of the device.[2]

[2]Reinitialization of the cache could be done here, but the kernel does not inform a block device driver when the block size changes.

B. Internals

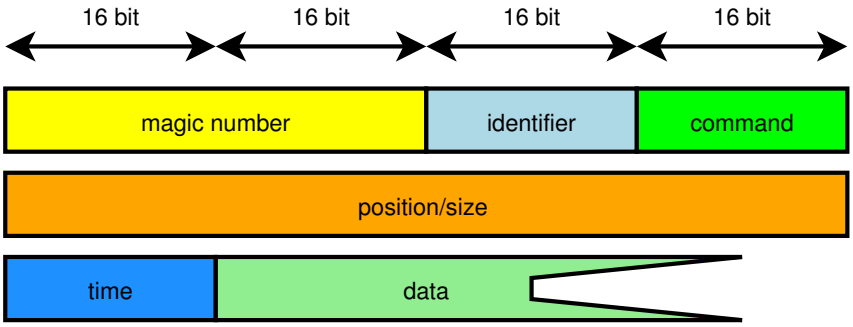
DNBD Protocol

Client Requests



- *Magic number*: The magic number is a constant (`0x19051979`), used to identify packets belonging to the DNBD protocol.
- *Identifier*: This field contains the identifier of the server which is queried. If the identifier is 0, all servers are addressed; this is used for heartbeats.
- *Command*: Only five of sixteen bits are used of the command field, to allow for further extensions. The two lowest bits define the command (`0x01` for handshake requests, `0x02` for read requests and `0x03` for heartbeat requests). The fourth bit (`0x08`) is set for client requests, the (`0x10`) fifth for server replies. Thus, a value of `0x0a` is a read request by the client.
- *Position*: For read requests, this field tells the server the position (in bytes) where it should read on the underlying block device or file. For handshake and heartbeat requests, the fields are undefined.
- *Time*: The time field corresponds to the internal timer of the kernel. This value is used for flow control.
- *Length*: The last field holds the amount of bytes requested by the client for read requests; otherwise it is undefined.

Server Replies



- *Magic number*: The first field contains the magic number of the DNBD protocol.
- *Identifier*: Here, a server inserts its unique identifier.
- *Command*: The two lowest bits are the same as in the corresponding request. The (`0x10`) fifth is set to mark the server reply.
- *Position/size*: For replies to read requests, the position field gives the position (in bytes) of the sent block. For handshake and heartbeat requests the fields contain the capacity of the device.
- *Time*: The time value is copied from the client request.
- *Data*: The data field contains the initial block size which the client should use for initialization; and if this is not the case, it carries the block itself.

C. Software and Documentation

Network Block Devices

Network Block Device (NBD)

`http://nbd.sourceforge.net/`

Original network block device by Pavel Machek, which found its way into the kernel.

Enhanced Network Block Device (ENBD)

`http://www.it.uc3m.es/ptb/~nbd/`

Peter T. Breuer's developed an enhanced version of NBD with failure handling and much more.

Another Block Device (ANBD)

`http://www.aros.net/~ldl/anbd/`

This is an implementation of a multi-threaded, but NBD compatible, client by Louis D. Langholtz.

Global Network Block Device (GNBD)

`http://sources.redhat.com/cluster/gnbd/`

The Global Filesystem uses its own network block device.

Diskless-related Software

Linux Diskless Clients

`http://www.ks.uni-freiburg.de/projekte/ldc/`

Framework and solutions to adapt standard Linux distributions for diskless use.

UnionFS

`http://www.fsl.cs.sunysb.edu/project-unionfs.html`

This stackable unification file system can merge a read-only file system with a writable one and provides write-semantics on diskless clients.

cowloop

`http://www.atconsultancy.nl/cowloop/`

Cowloop by Gerlof Langeveld offers copy-on-write semantics for block devices.

Network Simulation

Network Emulator (Netem) and Quality of Service (QoS) with iproute2

`http://linux-net.osdl.org`

The Network Emulator can establish network interfaces with delay, loss, jitter and other characteristics of unreliable communication. Other Quality of Service rules can limit bandwidth and prioritize specific traffic.

Documentation

The Linux Documentation Project

`http://www.tldp.org/`

A huge collection of manuals and tutorials concerning Linux and its abilities, e.g. IPv6, multicast, kernel programming and debugging, and much more.

List of Figures

Bibliography

[1] R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.*, 1:290–306, 1972.

[2] M. Beck et al. *Linux-Kernel-Programmierung.* Addison-Wesley, Bonn, Paris, 1995.

[3] P. T. Breuer, A. M. Lopez, and A. G. Ares. The Network Block Device. *Linux Journal*, (73), 2005.

[4] J. Corbet, A. Rubini, and G. Kroah-Hartmann. *Linux device drivers (3rd ed.).* O'Reilly Media, Inc., Sebastopol, CA, USA, 2005.

[5] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms (2nd ed.).* MIT Press, Cambridge, MA, USA, 2001.

[6] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems (3rd ed.): concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[7] H. Elaarag. Improving TCP performance over mobile networks. *ACM Comput. Surv.*, 34(3):357–374, 2002.

[8] D. Frascone. Debugging kernel modules with user-mode Linux. *Linux Journal*, 2002(97):5, 2002.

[9] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, 2000.

[10] M. Grimme and M. Hlawatschek. Aus dem Pool schöpfen. *iX-Magazin für professionelle Informationstechnik*, 2004(11):118–122, November 2004.

[11] H. Herold. *Linux/Unix Systemprogrammierung (3rd ed.).* Addison-Wesley, Munich, 2004.

[12] W. Jia, W. Tu, and L. Lin. Efficient Distributed Admission Control for Anycast Flows. In *ICCNMC '03: Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing*, page 78, Washington, DC, USA, 2003. IEEE Computer Society.

[13] R. Love. Kernel korner: I/O schedulers. *Linux Journal*, 2004(118):10, 2004.

[14] W. Mauerer. *Linux Kernelarchitektur*. Carl Hanser Verlag, Munich, Vienna, 2003.

[15] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen (4th ed.)*. Spektrum Akademischer Verlag, Heidelberg, Germany, 2002.

[16] D. von Suchodoletz and T. Zitterell. Hochstapler. *Linux Magazin*, (10):34–41, 2005.

[17] A. Wiebalck, P. T. Breuer, V. Lindenstruth, and T. M. Steinbeck. Fault-Tolerant Distributed Mass Storage for LHC Computing. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 266–275, Tokyo, Japan, May 2003. IEEE Computer Society Press.