RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arne Wiebalck

ClusterRAID: Architecture and Prototype of a Distributed Fault-Tolerant Mass Storage System for Clusters

Dissertation

HD-KIP-05-13

KIRCHHOFF-INSTITUT FÜR PHYSIK

INAUGURAL - DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von
Diplom–Physiker Arne Wiebalck
aus Bremerhaven

Tag der mündlichen Prüfung: 29. Juni 2005

ClusterRAID: Architecture and Prototype of a Distributed Fault-Tolerant Mass Storage System for Clusters

Gutachter: Prof. Dr. Volker Lindenstruth

Gutachter: Prof. Dr. Thomas Ludwig

ClusterRAID: Architektur und Prototyp eines verteilten fehlertoleranten Massenspeicher-Systems für Cluster

In den letzten Jahren haben sich Cluster aus Standard-Komponenten in vielen Bereichen als dominante Architektur für Hochleistungsrechner durchgesetzt. Wegen ihres besseren Preis-Leistungsverhältnisses haben diese Systeme, die typischerweise aus Standard-PCs oder Workstations und einem Verbindungsnetzwerk aufgebaut sind, die traditionell verwendeten, integrierten Supercomputer-Architekturen verdrängt. Aufgrund des zu beobachtenden Paradigmen-Wechsels von rein rechenintensiven hin zu Eingabe/Ausgabe-intensiven Anwendungen werden die in Clustern verwendeten Massenspeichersysteme zu einer immer wichtigeren Komponente. Daß sich bisher kein Standard für die Nutzung des verteilten Massenspeichers in Clustern durchsetzen konnte, ist vor allem der inhärenten Unzuverlässigkeit der zugrundeliegenden Komponenten zuzuschreiben.

Die vorliegende Arbeit beschreibt die Architektur und eine Prototypen-Implementierung eines verteilten, fehlertoleranten Massenspeichersystems für Cluster. Die grundlegende Idee der Architektur ist es, die lokale Festplatte eines Clusterknotens zuverlässig zu machen, ohne dabei die Schnittstelle für das Betriebssystem oder die Anwendung zu verändern. Hierbei werden fehler-korrigierende Codes eingesetzt, die es ermöglichen, die Anzahl der zu tolerierenden Fehler und somit die Zuverlässigkeit des Gesamtsystems einzustellen. Das Anordnungsschema für die Datenblöcke innerhalb des Systems berücksichtigt das Zugriffsverhalten einer ganzen Klasse von Applikationen und kann so die erforderlichen Netzwerkzugriffe auf ein Minimum reduzieren. Gründliche Messungen und Funktionstests des Prototypen, sowohl allein als auch im Zusammenwirken mit lokalen und verteilten Dateisystemen, belegen die Validität des Konzeptes.

ClusterRAID: Architecture and Prototype of a Distributed Fault-Tolerant Mass Storage System for Clusters

During the past few years clusters built from commodity off-the-shelf (COTS) components have emerged as the predominant supercomputer architecture. Typically comprising a collection of standard PCs or workstations and an interconnection network, they have replaced the traditionally used integrated systems due to their better price/performance ratio. As paradigms shift from mere computing intensive to I/O intensive applications, mass storage solutions for cluster installations become a more and more crucial aspect of these systems. The inherent unreliability of the underlying components is one of the reasons why no system has been established as a standard storage solution for clusters yet.

This thesis sets out the architecture and prototype implementation of a novel distributed mass storage system for commodity off-the-shelf clusters and addresses the issue of the unreliable constituent components. The key concept of the presented system is the conversion of the local hard disk drive of a cluster node into a reliable device while preserving the block device interface. By the deployment of sophisticated erasure-correcting codes, the system allows the adjustment of the number of tolerable failures and thus the overall reliability. In addition, the applied data layout considers the access behaviour of a broad range of applications and minimizes the number of required network transactions. Extensive measurements and functionality tests of the prototype, both stand-alone and in conjunction with local or distributed file systems, show the validity of the concept.

Contents

1. Introduction 15
- 1.1. Commodity Components in Clusters 16
  - 1.1.1. Hard Disk Drives 16
  - 1.1.2. Processors and Memory 17
  - 1.1.3. Networks and Busses 17
  - 1.1.4. Cluster Software 18
- 1.2. Requirements in Science and Industry 18
  - 1.2.1. High Energy Physics Experiments 19
  - 1.2.2. The Google Web Search Engine 19
- 1.3. Organization of the Thesis 20

2. Distributed Storage for Clusters 21
- 2.1. SAN, NAS, and RAID 21
- 2.2. Networked File Systems 24
  - 2.2.1. NFS 24
  - 2.2.2. DAFS 25
  - 2.2.3. MOSIX, MFS, and MOPI 26
  - 2.2.4. PVFS and Lustre 27
  - 2.2.5. GPFS 28
  - 2.2.6. Google File System 29
- 2.3. Integrated Mass Storage Systems 30
  - 2.3.1. HPSS 30
  - 2.3.2. CASTOR 31
  - 2.3.3. Storage Tank 32
- 2.4. Distributed Block Level Systems 33
  - 2.4.1. NBD and ENBD 34
  - 2.4.2. DRBD 36
  - 2.4.3. RAID-X OSM 37
  - 2.4.4. Petal 38
  - 2.4.5. Tertiary Disk 39
  - 2.4.6. DRAID 39
- 2.5. Distributed RAID 40
- 2.6. The need for something different 44

3. ClusterRAID Architecture 47
- 3.1. Design Considerations . . . 47
- 3.2. Overview and Functionality . . . 48
- 3.3. Performance . . . 56
- 3.4. Scalability . . . 62
- 3.5. Fault-tolerance . . . 63
  - 3.5.1. Basic definitions . . . 63
  - 3.5.2. Reliability Modeling . . . 65
  - 3.5.3. The Reliability of the ClusterRAID . . . 67

4. Error and Erasure Correcting Codes 73
- 4.1. Introduction . . . 73
- 4.2. Hamming Codes . . . 78
- 4.3. BCH Codes . . . 79
- 4.4. Reed-Solomon Codes . . . 80
- 4.5. Vandermonde-based Reed-Solomon Codes . . . 83
- 4.6. Other Algorithms . . . 86

5. ClusterRAID Implementation 87
- 5.1. Kernel Interfacing . . . 87
- 5.2. Overview . . . 90
  - 5.2.1. The Main Module . . . 92
  - 5.2.2. The Codec Module . . . 97
  - 5.2.3. The XOR Module . . . 102
- 5.3. Functional Analysis . . . 102
  - 5.3.1. Read Requests . . . 103
  - 5.3.2. Write Requests . . . 104
  - 5.3.3. Degraded Mode . . . 106
- 5.4. Additional Concepts . . . 109
  - 5.4.1. Locking . . . 109
  - 5.4.2. Copy Mode . . . 111
  - 5.4.3. Kernel Caches . . . 111
  - 5.4.4. Private Caches . . . 112
  - 5.4.5. Check Blocks . . . 112
- 5.5. Monitoring, Debugging, Control, and Configuration . . . 113
  - 5.5.1. Monitoring . . . 113
  - 5.5.2. Driver Control and Configuration . . . 115
  - 5.5.3. ClusterRAID Configuration and Startup . . . 116

6. Prototyping and Benchmarks 121
- 6.1. The Test Environment . . . 121
  - 6.1.1. The Hard Disks . . . 121
  - 6.1.2. The Network . . . 124
  - 6.1.3. The Network Block Device . . . 128

6.2. ClusterRAID Performance 128
  6.2.1. Local Testing 129
  6.2.2. Remote Testing 131
  6.2.3. System Tests 132
6.3. Functional Testing 140
  6.3.1. Online Repair and Node Replacement 141
  6.3.2. Stability and Consistency 141
  6.3.3. ClusterRAID and MFS 143
6.4. Implemented Optimizations and Possible Improvements 146
  6.4.1. Software 146
  6.4.2. Hardware Support 148

7. Conclusion and Outlook **151**

A. Appendix **153**
A.1. The Galois Field $GF(2^4)$ 153
A.2. A Production Quality Distributed RAID 153
A.3. The DWARW Module 155
A.4. The NetIO Network Benchmark 155
A.5. MOSIX related Measurements 156

Bibliography **159**

List of Figures

2.1. SAN schematic . . . 22
2.2. Comparison of conventional file access and DAFS . . . 26
2.3. GPFS configurations . . . 28
2.4. Google File System architecture . . . 30
2.5. HPSS architecture . . . 31
2.6. CASTOR architecture . . . 32
2.7. Storage Tank architecture . . . 33
2.8. ENBD architecture . . . 35
2.9. DRBD architecture . . . 36
2.10. RAID-X OSM data distribution . . . 38
2.11. Petal logical and physical view . . . 38
2.12. Distributed RAID architecture . . . 41
2.13. ENBD read throughput . . . 43
2.14. ENBD write throughput . . . 43
2.15. Distributed RAID functionality trace . . . 44

3.1. Functional overview of the ClusterRAID . . . 49
3.2. Data distribution scheme with redundancy nodes . . . 51
3.3. Data distribution scheme with distributed redundancy information . . . 52
3.4. ClusterRAID UML activity state diagram . . . 53
3.5. UML activity subdiagram for the reconstruction . . . 54
3.6. UML activity subdiagram for the write completion . . . 55
3.7. Concurrent throughput for dedicated redundancy nodes . . . 58
3.8. Maximum block write distribution . . . 60
3.9. Maximum block writes for distributed redundancy . . . 61
3.10. Expected ClusterRAID write throughput for distributed redundancy . . . 61
3.11. Schematic of a Weibull distribution . . . 65
3.12. ClusterRAID MTBF . . . 68

4.1. Basic elements of a data transmission system . . . 74

5.1. The kernel's structure `buffer_head` . . . 88
5.2. Schematic local view of a ClusterRAID node . . . 91
5.3. Schematic top level view of the ClusterRAID architecture . . . 91
5.4. ClusterRAID software architecture . . . 92

5.5. The ClusterRAID device type 94
5.6. The ClusterRAID disk type 96
5.7. The ClusterRAID type for meta buffer heads 96
5.8. The ClusterRAID type for buffer head containers 96
5.9. The ClusterRAID's structure for a redundancy algorithm 99
5.10. The structure `matrix_s` 99
5.11. The type `cr_rs_t` 100
5.12. UML sequence diagram of the ClusterRAID read path 103
5.13. UML sequence diagram of the ClusterRAID write path 104
5.14. Read path in degraded mode 107
5.15. Data structure passed to reconstruction 109
5.16. The structure `cr_block_lock` 110
5.17. Simplified split block write 113
5.18. The ClusterRAID driver output under `/proc/crstat` for an active data node 114
5.19. The ClusterRAID driver output under `/proc/crstat` for a redundancy node 115
5.20. Sample ClusterRAID configuration 117

6.1. Disk throughput for concurrent process pairs 123
6.2. Buffered disk throughput for concurrent process pairs on UP and SMP kernels 125
6.3. CPU load for concurrent process pairs 125
6.4. Benchmark of the used Fast Ethernet cards 126
6.5. Benchmark of the used Gigabit Ethernet cards 126
6.6. Latency measurements of the network cards used 127
6.7. ClusterRAID write throughput vs. the number of accessing processes 132
6.8. ClusterRAID read throughput for an (8,1) setup 134
6.9. ClusterRAID write throughput for an (8,1) setup 135
6.10. ClusterRAID CPU load for writes in an (8,1) setup 135
6.11. ClusterRAID write throughput in an (8,2) setup 136
6.12. ClusterRAID CPU load for writes in an (8,2) setup 136
6.13. Write throughput and CPU load for a variable number of redundancy nodes 137
6.14. Outgoing network traffic on a data node for a variable number of redundancy nodes. . 137
6.15. Comparison of the contributions to the throughput limitation 138
6.16. ClusterRAID throughput and load during reconstruction 139
6.17. ClusterRAID system test 142
6.18. MOSIX throughput improvement by process migration 144
6.19. ClusterRAID system test with MOSIX 145
6.20. Scaling behaviour of the Galois field multiplication with CPU clock speed 147

A.1. Distributed RAID prototype 154
A.2. Schematic view of the DWARW monitoring system 156
A.3. MOSIX process migration due to I/O load 157

List of Tables

2.1. The RAID Levels 24
2.2. ENBD seek times 42

3.1. ClusterRAID scalability 62
3.2. Reliability Parameters 70
3.3. ClusterRAID MTBF 70

4.1. The elements of $GF(2^3)$ 77

5.1. Thread-related functions of the `cr_main` module 98
5.2. Galois field methods of the codec module 101
5.3. Interface methods of the codec module 118
5.4. Commands of the ClusterRAID control library 119
5.5. Commands of the ClusterRAID launcher script 120

6.1. Bonnie++ measurement of the test rig disk with an ext2 file system 122
6.2. Streaming performance and latency of the test rig disk for direct device access 122
6.3. NBD performance measurements 129
6.4. NBD performance measurements in synchronous mode 129
6.5. ClusterRAID performance measurements for local access 130
6.6. Bonnie++ measurement of a ClusterRAID device 131
6.7. ClusterRAID performance for remote access 133
6.8. ClusterRAID performance during reconstruction 140

A.1. The elements of $GF(2^4)$ 153
A.2. Performance numbers of the production quality Distributed RAID 154
A.3. Available options of the netIO benchmark 157
A.4. MOSIX I/O Performance 158

1. Introduction

Work harder, work smarter, or get help.
Gregory Pfister

In June 1997 a new architecture debuted on the TOP500 list of the world's most powerful supercomputers [1]: with its 100 UltraSPARC-I processors the Berkeley Network of Workstations [2] was the first system labeled as *cluster* that entered the ranking. Since then, the share of clusters has steadily increased to about 60% in the current list. Thus, in the past few years clusters have replaced the classic architectures and are by now the predominant architecture for supercomputer installations.

The transition from monolithic to clustered systems since the early 1990s has mainly been triggered by the availability and the expeditious improvement of commodity hardware and software components, such as networks and microprocessors on the one hand and operating systems and programming tools on the other. These developments enabled commodity clusters to become a cost-effective alternative to classic supercomputers.

Although initially intended to satisfy the increasing demand for more processing power in parallel applications, clusters are nowadays faced with another transition, namely the shift from mere computing-intensive to data-intensive applications. Genome databases, geographic information systems, web archives and their information retrieval engines, or the upcoming high energy physics experiments are examples of potentially distributed applications, which combine the needs for capacities in the petabyte range with the demand for access rates exceeding several hundred megabytes per second. However, these size and bandwidth requirements can no longer be fulfilled by traditionally used, centralized storage solutions, as the available access speed of such systems is not sufficient for the size requirements of current applications. Storage architectures based on commodity components and more suitable for clusters are being developed and are beginning to replace the established storage systems.

An often neglected aspect in the design of commodity off-the-shelf clusters and in particular their mass storage services is the provision of fault-tolerance and reliability, which becomes necessary as the quality of mass market components is often worse than that of high-end products. This applies in particular to distributed disk-based storage systems, which try to close the gap between processing power and I/O performance. For local disk arrays, the RAID approach discussed in section 2.1 has become state of the art to protect from data loss by disk failure. This thesis presents the concept of a novel distributed architecture which extends the idea of RAID, namely to use a set of disks along with redundancy information to provide more data reliability, to the cluster level [3, 4, 5].

Before an outline of the thesis is given and in order to provide a basis for all later discussion, the technological background of cluster computing as well as the characteristics and requirements of two sample applications shall be examined.

1.1. Commodity Components in Clusters

The following discussion of the hardware and software components deployed in current commodity off-the-shelf clusters is confined to the fundamental aspects. As this thesis presents a disk-based mass storage architecture, special emphasis is laid on the status and trends in hard disk drive technology. More detailed surveys of the evolution of cluster systems and their constituent components may be found in [6, 7].

1.1.1. Hard Disk Drives

Since the introduction of the first hard disk drive in 1956 – IBM's famous RAMAC drive, having 5 megabytes capacity – the areal density of disk drives has improved by more than seven orders of magnitude. Areal densities of current disks are about 100 gigabit per square inch, and this value continues to double every year. That is, the growth rate of areal density even supersedes Moore's law [8], which is often quoted to emphasize the rapid developments in the computer industry, and it has outpaced both processor and communications technologies [9]. A number of breakthroughs – especially in head technology since the early 1990s – have led to this high compound growth rate. Currently, the technology of magnetic disk heads is based on the giant magnetoresistance (GMR) [10, 11]. As the heads move over the magnetic platters of the disk, the changes in the electrical resistance are measured to read the data from disk. Write elements use magnetic fields that alter the orientation of bit regions on the platter in order to store data.

The enormous growth rate in areal density has led to a substantial decline in hard drive prices. As areal density is doubled every 12 months, the price per unit disk capacity is approximately reduced by a factor of two in the same period of time. This trend substantiates their position as the leading device for mass storage.

As it is proportional to the *linear* density, the compound growth rate of the internal disk bandwidth is only 40% per year. Access and seek times even improve only at a rate of 10% per year. Hence, the gap between capacity and performance widens, and there are concerns that their ratio decreases to a level, where the overall system performance is affected adversely [12]. This situation may be exacerbated as new developments are on their way to overcome the physical limits set to GMR disk heads. Besides techniques to beat the superparamagnetic boundaries [13], there are several new approaches to further increase the number of bits stored per unit area, such as *Perpendicular Disks* [14], *Heat-Assisted Magnetic Recording* [15], or *Patterned Media* [16] to name only a few.

The expected lifetime of disk drives, measured by the mean time between failure (MTBF), also improved over the past few years, but not at the same pace as capacity. According to disk manufacturers, the MTBF of a disk is around 1,000,000 power-on hours. However, field experience shows that in real environments the value can be significantly lower [17]. Another important measure is the bit error rate, i.e. the rate at which bits cannot be retrieved correctly from the device. Bit error rates of current disks are in the range of 10^{-15}, a value that has not changed significantly over the last ten years. Due to the size of installations with thousands of disks and their corresponding online disk capacity in the petabyte range, failing devices and bit errors will become a major issue that higher system-level services will need to take care of.

Aside from the aforementioned developments in storage media and head technology, there are also trends to exploit the processing power of the disk for more advanced tasks. Recent projects, such as

Active Disks [18, 19, 20] or the *Multi-view Storage System (MVSS)* [21], seek to source out certain aspects of data management into the device itself in order to improve the overall system performance and to relieve the host processor.

Although a slowdown in progress is projected due to the significant technical challenges and technological changes, there is nevertheless a widely-held view that disks and disk-based systems will continue to be the prevalent mass storage technology for at least another ten years.

1.1.2. Processors and Memory

Processor clock speeds and memory sizes closely follow the aforementioned Moore's law, i.e. they double every 18 months. The time required for memory access, however, is improving at less than 10% per year [22]. Similar to the growth of disk capacity and its internal bandwidth or the need of applications for petabyte storage and the problems of centralized solutions to provide reasonable access rates, current clock frequencies overstrain memory access speed. The introduction of faster intermediary memories as caches mitigates this situation to a certain degree, but the gap is still widening every year. Besides physical limits, which may hinder the ability of clock speed to continue to increase at the same rate, power dissipation and CPU cooling is becoming a major issue. This has led to a shift from mere clock rate tuning to feature enhancements of new processors. Extensions of the instruction set, such as MMX [23], SSE [24, 25], AMD64 [26], or EM64T [27], are accompanied by new approaches, such as simultaneous multithreading (two or more virtual processors) and dual core designs (second real processor on the same chip).

Although 64-bit and 64-bit extended 32-bit processors have been available for several years, 32-bit x86-based CPUs still dominate the PC market. Based on the results of the Top500 list, most of the current supercomputer installations also rely on 32-bit based rather than on 64-bit based processors. A reason for this may be the lack of 64-bit applications and the marginal performance improvement due to the internal use of 128-bit words in extended 32-bit processors.

1.1.3. Networks and Busses

The Intel-created [28] Peripheral Component Interconnect (PCI) [29] and its extension PCI-X [30] are currently the predominant busses in the commodity market. However, they may be replaced in the near future by the newly developed PCI Express (PCIe) [31] interconnect. While PCI and PCI-X are real busses, PCIe establishes switched point-to-point connections between its end devices, each of which can use the full bandwidth of up to 4 GB/s for PCIe x16. This bandwidth is absolutely comparable to contemporary memory bandwidths.

Clusters usually deploy low-latency, high-bandwidth interconnects when it is necessary to reduce the impact of limited internode communication bandwidth and latency on the overall system performance. Well established high performance network technologies which provide latencies in the microseconds range and bandwidths of up to several 100 MB/s include Myrinet [32], Quadrics [33], or the Scalable Coherent Interface (SCI) [34]. InfiniBand [35] is already gaining popularity and may also take a leading role in the interconnect area.

If the requirements on the interconnect – especially in terms of latency – are not that strict or if an additional control network is useful, Ethernet [36] provides a low cost alternative due to its dominating role in the PC mass market. As Ethernet is practically always used together with the TCP/IP protocol [37], not only latency but also CPU overhead is a major drawback compared to the high performance networks due to the complexity of the TCP/IP protocol stack. Both drawbacks can be mitigated by the usage of more specialized protocols, such as GAMMA [38] to reduce latency or STP [39] to reduce CPU overhead. Alternatively, the deployment of specialized hardware, such as TCP offload engines, is an option to overcome the drawbacks of Ethernet and TCP/IP.

1.1.4. Cluster Software

Operating systems deployed to run commodity off-the-shelf cluster installations do not reflect market shares to the extent hardware does. On the contrary, the development of the GNU/Linux [40, 41] operating system in the early 1990s and the steadily increasing improvement and support by a large community was a major driving force behind the success of PC clusters. Aside from the fact that the scientific community is traditionally more involved with UNIX-like operating systems, GNU/Linux has a number of advantages that makes it particularly attractive to be used in clusters, compared to other solutions. GNU/Linux is freely available from the Internet, it runs on and with almost all available hardware, it provides the necessary stability and reliability, bugs are usually fixed quickly, and its open source approach allows fine-grained tuning and adaptation if necessary. Although approaches exist that mask a cluster as a single system instead of a collection of independent nodes, such as the GNU/Linux extension MOSIX, described in section 2.2.3, or SCore [42], these solutions are less popular than that provided by GNU/Linux.

Message passing libraries and distributed shared memory (DSM) systems are the two major approaches for communication of parallel programs running on different nodes within a cluster. Message Passing Libraries provide interfaces and routines for parallel applications to establish communication based on explicit message exchange. Popular systems include the implementations of the Message Passing Interface (MPI) standard [43], for instance MPICH [44] or LAM/MPI [45], as well as the Parallel Virtual Machine (PVM) [46]. Distributed shared memory systems transparently offer the user a physically distributed but logically shared memory. This can be realized in hard- or software. DSM systems, however, are much less wide-spread as the deployment of message passing systems.

A number of parallel debuggers and profilers, performance analyzers, or monitoring and administration tools have been developed to account for the increasing use of clusters. A discussion of all these tools, however, is beyond the scope of this thesis.

1.2. Requirements in Science and Industry

Two applications – one from science and one from industry – shall be briefly examined in order to illustrate sample requirements on mass storage in current cluster-based applications. The main foci are the needed capacity, the measures to protect against data loss, and the access behaviour of the corresponding applications.

1.2.1. High Energy Physics Experiments

The total amount of data produced by the upcoming high-energy physics experiments at the Large Hadron Collider (LHC) [47] at the European Laboratory for Particle Physics (CERN) [48] will be about 12 to 14 petabytes per year. The experiments will deploy large PC clusters as part of the trigger and read-out chain, for online data processing, or for offline data analysis. However, in order to analyze this massive amount of data, a hierarchy of off-site computing centers ranging from very large multi-service facilities to large PC farms, which provide special purpose services, is being established [49]. This hierarchy of computing resources, called *Regional Centers*, comprises five levels of complexity and capability, referred to as *Tiers*. There will be only one Tier-0 center, located at CERN. The other regional centers are geographically distributed and interconnected by fast networks. Typical Tier-1 regional centers will deploy farms of commodity off-the-shelf computers and provide services such as event reconstruction and data analysis. The ATLAS [50] experiment, for instance, will require approximately 1,000 twin CPU nodes for one of these Tier-1 centers [51].

The actual data analysis for these experiments is a complex multi-stage process and will therefore be presented here in a simplified fashion. The numbers of the ATLAS experiment will serve as an example to identify the crucial aspects in this analysis process [52].

Every year the ATLAS experiment will record the data of 2.7 billion particle collisions, the so-called *events*, each of which records will be 2 megabytes in size. Hence, the raw data produced by the ATLAS experiment alone will be approximately 5.4 petabytes per year. This raw data will be stored at CERN. After a first processing step, in which simple data reconstruction is done, the resulting *Event Summary Data (ESD)* is sent out to the tier centers for further analysis. The space required to store these ESD is between 5 and 10% of the space required for the raw data. Within the tier centers the data is reprocessed several times and passes through a number of data types. The important thing to note here is that each of these processing phases may be repeated several times and that the derived data type is always significantly smaller in size than the previous one. Hence, the ratio of reading to writing is larger than 1. Moreover, the actual data analysis is easily parallelizable as it operates on the granularity of events. Since these events are completely independent from each other, the analysis processes can simultaneously work on independent data sets.

Due to the tremendous costs of operating the collider, the collected data must be protected against data loss. This protection is provided by replication directly at CERN and by the distribution of data to the tier centers. Although all data used below the Tier-0 level is a derivative of the raw data, a repeated transfer and the subsequent reprocessing costs make data reliability on the Tier-1 level mandatory. Usually, this is taken care of by the deployment of tape-based backing stores.

1.2.2. The Google Web Search Engine

As of today, Google [53] is one of the world's most popular web search engines indexing more than 8,000,000,000 web pages. Google's internal architecture is based on clusters with commodity-class PCs [54]. The clusters are geographically distributed over the world in order to protect the service against disastrous data center failures, such as fire or earthquake. In addition, this approach provides a quick turn-around time and the possibility of load balancing by routing user requests to the physically nearest cluster. Each of the clusters comprises about 1,000 standard PCs running a GNU/Linux operating system. Once a search request has been sent to one of the clusters, the subsequent processing consists of two main phases. From the user's query words an index is consulted to find a set of relevant documents for each keyword. The intersection of these document lists for all query words is the result of the first processing phase. In the second phase, the document identifiers are used to assemble a result page containing a uniform resource locator and a document summary. Fortunately, this kind of processing is amenable to a comprehensive parallelization, as the index can be divided into smaller index sets, each of which is processed simultaneously. The subsequent merging step is relatively cheap. Beyond the optimal price-performance ratio for this particular application, the architectural choice to use a cluster of mid-range PCs instead of a smaller number of high-end servers facilitates the increase in the number of processing steps per query, the deployment of more complex algorithms, and accommodates index growth.

Typical queries require the system to read several hundred megabytes from the disks of the cluster nodes. Updates are comparatively rare, i.e. the number of read accesses is much larger than the number of writes. Google's in-house developed distributed storage solution, the Google File System discussed in section 2.2.6, accounts for this particular access behaviour.

Availability and reliability inside Google is provided by replication. This holds true for both services and data. Faulty index or document servers are excluded from the cluster-internal load balancing system. However, all services remain available and the capacity is only reduced by the fraction the unavailable machine represents of the total capacity. Maintaining several copies of the data provides sufficient protection against data loss and eases the deployment of load balancing, but it induces a significant space overhead. Since Google maintains three replicas of their data within a cluster, only a third of the whole capacity is usable.

As of today, applications that require storage in the petabyte range and that deploy thousands of servers are only beginning to emerge. However, the intrinsic traits of the HEP experiments and Google as described above, such as a high degree of request-level parallelism, the requirement for data reliability, the constraint of an optimal price-performance ratio, or the bias to read accesses, are shared by many other applications. The system presented in this thesis thus attempts to account for these requirements.

1.3. Organization of the Thesis

This thesis is organized as follows. After a survey of the variety of distributed storage solutions in chapter 2, the ClusterRAID architecture is presented in chapter 3. The discussion sets out the architectural principles and covers issues such as the expected performance, scalability, and reliability. Chapter 4 gives an introduction into the theory of error and erasure correcting codes and presents algorithms suitable for a ClusterRAID implementation. A detailed discussion of the design and implementation of a ClusterRAID prototype as an extension of the GNU/Linux operating system is the focus of chapter 5. Prototyping and benchmarks results are presented in chapter 6, before the thesis is closed by a final conclusion and outlook for further research.

2. Distributed Storage for Clusters

Standard
One of the most complex German words.

As of today, there is no state-of-the-art architecture for distributed reliable storage in PC farms. This is in particular true for larger installations with hundreds or thousands of nodes. Centralized file servers with all their drawbacks regarding performance, fault-tolerance, and scalability, connected via a network file system may be the most wide-spread approach. Often chosen for its ease of installation, this kind of architecture may not be sufficient, however, if there is a demand for low latency data access, high throughput, high availability, or long term durability. Some of the current approaches used to address these issues will be outlined in this chapter. The spectrum of different requirements for secondary storage is reflected in the number of architectures presented. Storage networks, network file systems, shared disk file systems, integrated mass storage systems, and distributed block level systems will be discussed in terms of their aims, advantages, and drawbacks. Since the ClusterRAID is a storage architecture for clusters, a discussion of wide-area network storage or file systems such as Oceanstore [55], CFS [56], or PAST [57] is omitted.

2.1. SAN, NAS, and RAID

In centralized architectures mass storage is only accessible via servers. Disks, disk subsystems, or tapes are generally built-in components and access to data is only possible via the hosting machine. This approach gives rise to a number of problems, such as scalability, storage consolidation and waste of capacity, the time required to deploy new storage, data availability, or the sharing of data among clients. Some of these problems apply more to single server installations, the others are more likely to occur in multi-server sites. Storage Area Networks (SANs) and Network Attached Storage (NAS) can help to reduce these problems [58].

In a SAN, the cables connecting a server and the storage media are replaced by a network, see figure 2.1. This approach decouples servers and storage devices. Via a SAN file system, which takes care of data consistency and conflict resolution, several servers have direct access to the shared devices. No interaction with other servers is necessary. From a data path's perspective the SAN interface is similar to a disk interface: SAN devices typically appear as block oriented SCSI devices, but the SCSI commands are transferred over the network to the storage devices.

The decoupling of servers and storage reduces the waste of capacity, as the servers now have access to all storage devices. The storage capacity is consolidated in a pool of shared devices. In addition, since storage devices are not associated with a specific server, it is possible to exchange system components during client operation. Data availability and system reliability can be increased by implementing redundant I/O paths. Backups of the data can be carried out when devices are idle,

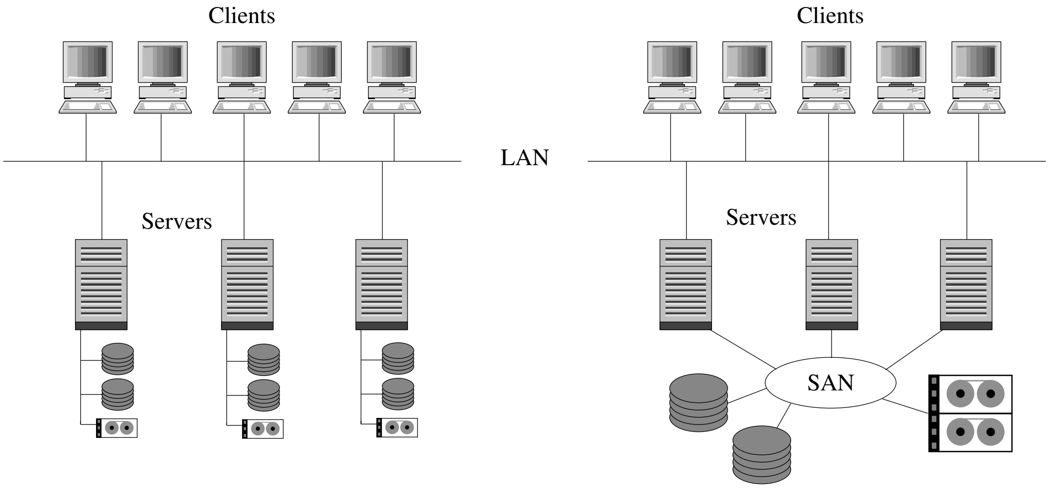

FIGURE 2.1.: Comparison of a centralized architecture (left) and a Storage Area Network (right). The SAN replaces the cables connecting the server to the storage devices by a network.

without active server involvement.

However, a SAN is more complex than a single server. The system consist of more components that can fail. Data is transferred over a network resulting in more processor overhead compared to local device access. In most cases however, these drawbacks are acceptable considering the advantages of networked storage.

Nowadays, the technology used in most SAN installations is Fibre Channel (FC), together with the Fibre Channel Protocol (FCP) [59]. However, due to its increasing performance and its decreasing cost Ethernet in conjunction with IP-based protocols, such as iSCSI [60], iFCP [61], FCIP, or mFCP [62], is becoming an interesting alternative. All the mentioned protocols transfer SCSI commands over IP and thus, in most cases, Ethernet. Internet SCSI (iSCSI) is a SAN protocol, where Ethernet and TCP/IP replace FibreChannel and FCP. An iSCSI driver maps the SCSI daisy chain on a TCP/IP network. Currently, the server CPU runs the protocol. However, iSCSI host bus adapters will relieve the CPU in future implementations. HyperSCSI [63], which avoids the TCP/IP stack by transferring the commands in raw Ethernet packets, or iSCSI implementations over UDP may also help to reduce the CPU load. The iSCSI protocol is being standardized by the Internet Engineering Task Force (IETF) [64, 65] and the first iSCSI devices are now available. Internet FCP (iFCP) attempts to save the investment in FibreChannel devices and is a port of FCP to TCP/IP. The FibreChannel network can thus be replaced by an IP network. FibreChannel over IP (FCIP) is a tunneling protocol connecting FibreChannel SANs over large distances using an IP network. To improve the performance in environments with small amounts of transmission errors Metro FCP (mFCP) uses UDP instead of TCP. In addition to these IP-based protocols, the serial InfiniBand interconnect is also a candidate to replace FibreChannel in the near future.

Another widespread approach for networked data storage are NAS systems. Typically, these are storage appliances configured as file servers with an operating system optimized for file sharing. Via a LAN connection the internal storage capacity is provided to the clients. The difference to a SAN is the client interface. NAS systems provide a file system interface, data is served using a network file system such as NFS, see section 2.2.1. In most cases, the network technology for NAS systems is Ethernet. NAS servers do not necessarily have disks directly attached to them: NAS gateways provide access to a SAN they are built upon. In such installations a NAS inherits the advantages and drawbacks of a SAN mentioned above.

In order to address reliability problems with large single disks as well as to improve the I/O performance, the concept of RAID (Redundant Array of Inexpensive/Independent Disks) [66] has been proposed. It has been widely adopted over the past years, in particular in NAS systems. The cited publication originally detailed five strategies, usually referred to as *RAID levels* and numbered from 1 to 5, which use different strategies to protect against data loss, to improve the performance, or both. Later on, additional RAID levels have been developed, among them linear RAID, RAID-0, RAID-6, RAID-10, or RAID-53, to name the most common ones. The basic idea of all RAIDs is to distribute the data over several devices. One strategy used by several RAID levels is *striping*: logically consecutive data is subdivided into blocks, which are stored in a round-robin fashion on different disks. This improves the transfer rate in streaming mode as multiple requests can be issued to the constituent devices in parallel. The actual RAID level determines, *how* redundant information is generated, if at all, and how the data and the redundancy information is spread over the devices.
Linear RAID is a simple concatenation of several disks into one large virtual device. It provides no protection against device failures and the performance does not differ from that of a single disk. *RAID-0* uses all its constituent devices for data striping. As in linear RAID, no redundancy algorithm protects from data loss. However, due to the application of striping it provides better transfer rates. *RAID-1* duplicates data to a clone disk. Hence, in case of a disk failure, the data is still available by means of the other disk. Read accesses can also be sped up by reading the data in a stripe fashion. *RAID-2* uses Hamming codes, which are introduced in section 4.2, to protect from data loss. At a comparable performance it needs more space than *RAID-3* and is hence rarely implemented. The latter RAID level stripes data on a byte level. As all disks are accessed in every I/O operation, RAID-3 delivers a high transfer rate, but works poorly for applications with high request rates. Data protection is achieved by parity storage on a dedicated device. *RAID-4* and *RAID-5* stripe on block level and access each disk individually. While RAID-4 uses a dedicated parity device, which can easily become a bottleneck, RAID-5 distributes the parity information over all underlying devices for load balancing purposes. As RAID-5 is a compromise between performance and reliability, it is very widely used. *RAID-6* is a set of redundancy algorithms that protect against simultaneous loss of two devices. For instance, in the so-called *P+Q redundancy* two independent checksums are used, in *2-dimensional parity* the devices are logically organized in a two-dimensional grid, wherein parity is computed horizontally over the rows and vertically over the columns. RAID-6 is the only RAID level that can tolerate all combinations of double device failures. The remaining levels are combinations of the previous ones. *RAID-10* uses striping over mirrored disks. Although this approach uses only half of the total capacity, it combines the performance gain of data striping with the reliability of mirrored disks. RAID-10 can tolerate more than single disk failures in an array, if the faulty disks do not belong to the same underlying RAID-1 set. If the underlying disks shall contain more than two devices, *RAID-53* is the RAID level of choice. While the upper RAID-0[1] performs good at high

---

[1] Following the nomenclature for RAID-10, one would expect RAID-53 to be a RAID-3 array of RAID-5 devices. Sur-

RAID level	Redundancy algorithm	Space usage	Tolerable failures
linear	None	N	0
0	None	N	0
1	Mirroring	2N	1
2	Hamming Codes	≈1.5N	1
3	Parity	N+1	1
4	Parity	N+1	1
5	Parity	N+1	1
6	P+Q or 2-dim. Parity	N+2 or MN+N+M	2
10	Mirroring	2N	1
53	Parity	N+stripe factor	1

TABLE 2.1.: Overview of the different RAID levels. The space usage denotes the total needed capacity, where N (or NM for RAID-6) is the amount of disks available to store non-redundant user data. The tolerable failures denote the maximum number of faulty devices a RAID level can get along with for all possible device combinations.

request rates, the underlying RAID-3 offers high transfer rates. Hence, RAID-53 is a compromise with good overall performance.

As a summary, table 2.1 provides an overview of the original and the newly added RAID levels. A more detailed discussion of the advantages and shortcomings of the different RAID levels can be found in [67].

2.2. Networked File Systems

Networked or distributed file systems are the extension of local file systems to distributed environments: a network connection is now part of the data path between the user process and the storage media. As already mentioned in the introduction, there are many demands on distributed mass storage systems and networked file systems. This fact has led to a huge number of different approaches, architectures, and implementations, and only a short overview of a small representative selection of them will be provided in this section. Since Sun's Network File System (NFS) is the most widely used networked file system, it will be presented first. All other systems have been selected because they lay special emphasis on one or more of the requirements that distributed mass storage systems must deal with. Among them are the design goal of minimized CPU load (DAFS), minimized network load (MFS), throughput and scalability (PVFS and Lustre), the support for parallel access (GPFS), or the ability to provide reliable mass storage on inherently unreliable components (Google FS).

2.2.1. NFS

Developed by Sun Microsystems in 1985 NFS [68] has become the most widespread distributed file system today. There are several reasons for its popularity. NFS follows a simple architectural approach: clients can import directories from a server and mount them into their local file system. It

---

prisingly enough, it is a RAID-0 array of RAID-3 devices.

hides the fact that the files are remote and imported over the net so that clients can access files as if they were local. The architecture is quite simple, and so is the administrative overhead. Servers specify which directories are exported and which clients are allowed to access them. The clients simply specify the server, the directory they wish to import, and the local mount point. A second strength of NFS is its hardware independence and thus its support for heterogeneous environments. NFS uses the *eXternal Data Representation* (XDR) [69] data format for communication between server and client. This protocol is machine independent and thus allows client and server to run different operating systems. In addition, it makes NFS independent from the underlying communication mechanisms.

One of the design goals of NFS was to provide simple failure recovery mechanisms. NFS achieves that by its statelessness. An NFS server does not have any information about open client connections. All information needed for data transfer is kept inside both the requests and replies exchanged between client and server. If a client or a server fails, no information about files or their data is lost.

NFS improves its performance by caching on both client and server side. On the server side, accesses to the disk are minimized if the requested data is already in the cache. On the client side, a request is not sent to a server if it can be served from the local cache. The introduction of client-side caching inevitably introduces the problem of coherency. If two clients write simultaneously to their cached copy of the same file, the result is unpredictable, just as it is for two programs writing to a local file. NFS tries to alleviate the problem by cache ageing. If a data block has been in the cache for longer than a certain timeout, usually several seconds, it is deleted from the cache and must be requested from the server again upon the next access. The opening of a file in the cache also triggers a server request for its modification time. If the file has been modified by another client while it was cached, the copy is deleted and the actual version is requested from the server. Finally, all metadata information is synchronized with the server every 30 seconds. All these actions, however, do not prevent data inconsistencies, they only make them less probable.

An additional drawback of NFS is its lack of scalability. Both fault-tolerance and scalability are limited by the centralized server approach. If the server fails, the data is not available for client access. Although failover servers can be set up to take over if the primary one fails, the problem of data consistency among the different servers arises. The scalability problem is even more severe. If a large number of clients are connected to a single NFS server they all share the interconnect bandwidth. NFS is surely not the file system of choice in situations where hundreds of clients in a cluster are running I/O intensive applications.

2.2.2. DAFS

The Direct Access File System (DAFS) [70] is a shared-file access protocol using the Virtual Interface Architecture (VI) [71]. A key feature of the VI Architecture is that it allows applications to queue data transfers directly to VI-capable hardware, without operating system intervention. This avoids both the context switch overhead and the overhead induced by copying data between buffers inside the operating system. Since packets are assembled and disassembled by VI-compliant hardware, which also takes care of message fragmentation or correct alignment, the usual network overhead can also be reduced. These two features allow an efficient transfer of data between communicating machines, reflected in a low CPU overhead and a reduced latency.

Figure 2.2 shows the difference in the data path between conventional file access and DAFS. For

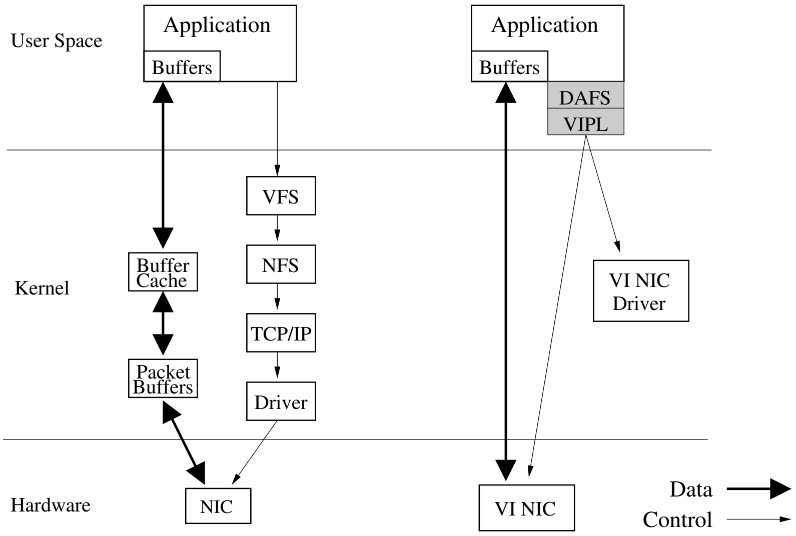

FIGURE 2.2.: Comparison of the data and control paths for conventional network file access and for DAFS.

standard network file systems, data is copied from application buffers to the file system cache or to the buffer cache of the underlying device. Another copy step from this cache to packet buffers is performed, before control is transferred to the network interface card. Unless specialized hardware is used, the network protocol is completely handled in software. Contrary to that, the data in DAFS is directly moved from application buffers to VI-capable hardware. The application side interface to DAFS is provided by a user space library, the DAFS file access library. This library requires a VI provider library (VIPL) in order to communicate with the NIC and its driver.

DAFS thus allows direct access from application clients to shared file servers. Data consistency is guaranteed by record locking. Clients require a lock in order to be allowed to modify a data record. Locks can be cached to improve performance in case of repeated accesses to the same data. DAFS provides built-in fault-tolerance and can deal with both client and server failures. Since VI is the transport mechanism for DAFS, the network technologies that can be used with DAFS are the same for which VI implementations exist. Among these are FibreChannel, InfiniBand, and Ethernet.

2.2.3. MOSIX, MFS, and MOPI

MOSIX [72] is a distributed operating system that supports preemptive process migration for load-balancing purposes. It is able to monitor the resources consumed by a process, such as CPU load or memory usage, and uses this information to transfer processes from one node to another in order to balance the overall load. System calls of migrated processes are intercepted by the so-called link layer on the remote node. If the call is node-independent, it is served on the remote node. If it is not, the system call is forwarded to the home node of the process.

The distributed MOSIX File System (MFS) and the Direct File System Access (DFSA) [73] have been developed to reduce the overhead in the case of I/O bound applications. MFS provides a consistent view on all files on the nodes in a MOSIX cluster. This enables elegant access to data on remote nodes. Additionally, the DFSA, a re-routing mechanism, further reduces the overhead imposed by

I/O bound processes. None the less, I/O activities of processes are also monitored and may lead to a positive migration decision. Thus, I/O intensive processes are moved to their data, instead of the traditional approach of moving the data to the process.[2]

The MOSIX Parallel I/O System (MOPI) [74] is a user space library that provides parallel access to data distributed over several nodes. Data is separated into so-called data segments, which are distributed over the contributing nodes. A meta unit keeps track of the segments belonging to one file, while a metadata manager grants access to segments on remote nodes.

Performance tests demonstrate the scalability of the MOSIX I/O system. By migrating I/O intensive applications, the aggregate throughput is close to the aggregate local disk bandwidth. The approach of moving the application to its data reduces the network load significantly compared to the centralized server approach. However, access to data using MOPI is only possible by means of a user library, i.e. applications have to replace their system calls by the corresponding MOPI calls. Furthermore, as of now, there is no fault-tolerance provided by MOSIX, which will become a problem when the system is installed on larger clusters.

2.2.4. PVFS and Lustre

The Parallel Virtual File System (PVFS) [75] is a production-quality high-performance parallel file system for Linux clusters. Its architecture follows the client-server paradigm using multiple servers, the so-called I/O daemons. These daemons usually run on I/O nodes, which are special nodes in the cluster dedicated to I/O. User applications run on compute nodes, which need not to be distinct from I/O nodes. A metadata manager handles all file metadata information. This manager is contacted by user processes for operations such as open, close, create, or remove. The metadata manager returns information that is used by the clients to contact the I/O daemons for direct file access. PVFS stores data on local file systems and not directly on block devices. This holds true for both user data and metadata. This approach enables PVFS to retain a user-level implementation[3].

PVFS has several user interfaces: applications can use the UNIX/Posix [76] interface, a PVFS native API or MPI-IO [77]. It provides a cluster-wide consistent name space for the files. An interesting feature of PVFS is the possibility of user-controlled data striping. Applications can decide how many I/O nodes a file is to be stored on. In addition, the stripe size can also be set by client processes. This way, an application has control over the parallelism during file access. Partitioning is another feature of PVFS allowing for access of multiple non-contiguous regions in a file by a single system call. In order to provide the UNIX/Posix API, system calls are intercepted and replaced by patched versions. An open call for a PVFS file includes the contact with the metadata manager, for instance. This, however, forces the PVFS system to be changed with the system libraries. Since PVFS has no built-in mechanisms for fault-tolerance, attempts are being made to enhance PVFS with fault-tolerance features [78, 79].

Lustre [80] aims to provide a scalable, high-performance file system for Linux clusters. Similar to PVFS, the architecture comprises a metadata server, which is contacted in case of file system

---

[2] As will be shown later in this thesis, the ClusterRAID architecture follows a similar approach, i.e. to keep local as many accesses as possible. As the data accessed by an application is in many cases larger than the application itself, and since the amount of data to be processed increases for most applications, the approach of moving the application and leaving the data in place is reasonable.

[3] A kernel-based implementation of PVFS also exists.

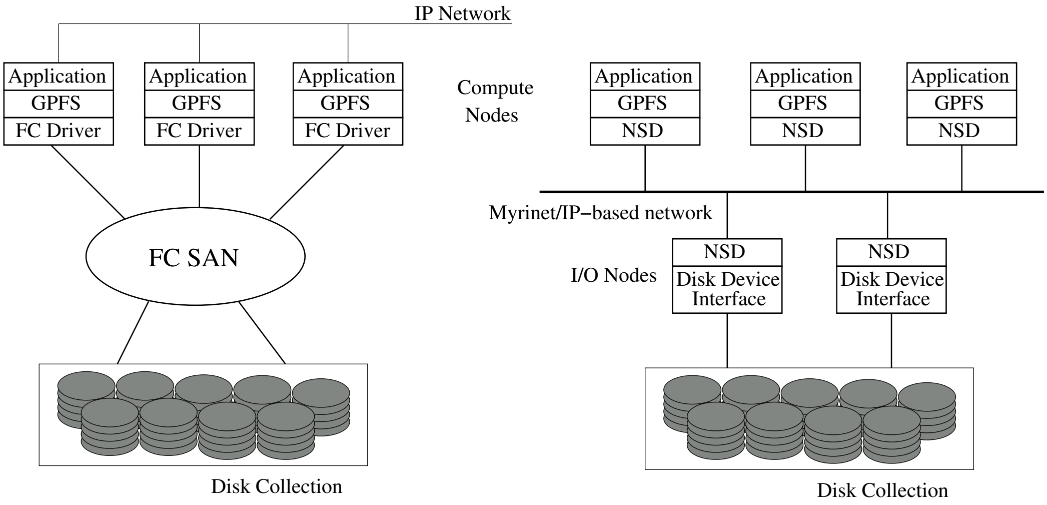
IP Network
Application
GPFS
FC Driver
Application
GPFS
FC Driver
Application
GPFS
FC Driver
FC SAN
Disk Collection
Compute Nodes
Application
GPFS
NSD
Application
GPFS
NSD
Application
GPFS
NSD
Myrinet/IP–based network
I/O Nodes
NSD
Disk Device Interface
NSD
Disk Device Interface
Disk Collection

FIGURE 2.3.: The two basic GPFS configurations. Left: The Directly Attached Model. Right: The Network Shared Disk Model. The image is taken from [82].

metadata changes. The actual I/O is again performed only between the client node and the servers that hold the data. In Lustre's terminology, the servers are called Object Storage Targets (OSTs). They manage the data stored on the underlying devices, the so-called Object Based Disks (OBDs). This two-fold storage abstraction allows for an easy integration of smart storage devices with object-oriented allocation and data management in hardware and follows the general trend of offloading I/O to the devices itself. In addition, the OST abstraction layer provides a flexible way to add new storage to existing installations. The use of a network abstraction layer allows for an easy port of Lustre to several networks. Implementations exist or are planned for TCP, Quadrics, Myrinet, FibreChannel, and InfiniBand. Lustre provides built-in fault tolerance: replicated, failover metadata servers and distributed OSTs increase the file system availability by eliminating single points of failure. As of today, the implementation of Lustre is not yet completed. In the future Lustre's performance is planned to be enhanced by the implementation of sophisticated caching strategies and the use of distributed metadata servers in order to avoid bottlenecks for larger installations.

2.2.5. GPFS

IBM's General Parallel File System (GPFS) [81] is a shared disk file system designed for concurrent high-speed file access from any node in a cluster. GPFS can be deployed in two basic configurations, which are depicted schematically in figure 2.3. In the first configuration, referred to as the *Directly Attached Model*, all nodes in a cluster are directly connected to a disk collection via a FibreChannel SAN. Applications access data by means of a GPFS driver. In environments where not all nodes have direct access to the SAN, the *Network Shared Disk (NSD)* model of GPFS is preferred. The cluster nodes are then divided into two classes. NSD servers have direct access to the GPFS disks. On these servers, the NSD driver, a software abstraction layer, provides the abstraction of disk blocks to the other class, the compute nodes. The compute nodes are attached to the NSD servers by an IP-based network or a high-speed interconnect, such as Myrinet. On the compute nodes, the NSD

driver emulates a SAN. Thus, from the client-side GPFS looks just like in the first configuration.
Since GPFS is a shared disk file system, it must provide some locking mechanisms to protect from data corruption through concurrent accesses. This is achieved by a token-based distributed lock manager working on block level. Additional support for parallel access in GPFS is realized by extended interfaces to be used in case of performance reduction due to difficult access patterns. GPFS can recognize typical access patterns and improves the throughput by sophisticated pre-fetching mechanisms.
In contrast with other distributed file systems, GPFS does not require a central metadata server. Instead, metadata is handled at the node which is using the file. This approach avoids the central server becoming a bottleneck in metadata intensive applications, and it also eliminates this server as a possible single point of failure. Since it uses the Reliable Scalable Cluster Technology (RSCT) [83] providing a heartbeat and failure reporting capability within a cluster, GPFS can also recover from node, disk connection, or adapter failures. If multi-tailed disks are used, they can be connected to multiple I/O servers providing both load balancing and redundant paths to the data. Protection from loss of data and metadata in GPFS is provided by replication.

2.2.6. Google File System

The design of the Google File System (GFS) [84] has been heavily influenced by Google's workload and technical environment. As outlined in the introduction, Google's web search engine is built from commodity-class PC farms, each of which consists of the order of 1,000 nodes. Therefore, monitoring, error detection, fault-tolerance, and automatic recovery must be integral parts of a distributed file system used by these machines. Typical Google files are several gigabytes in size, which influenced file system parameters, such as the block size or the implementation of I/O operations. Google's files are written once and then read multiple times, usually in a sequential manner. Taking the typical file sizes into account, client-side caching is not the method of choice. A schematic overview of GFS is depicted in figure 2.4. Data is stored in chunks, ordinary Linux files of typically 64 megabytes, which are stored on the disks of the chunk servers. Just as in PVFS, the chunk servers use the local file system to buffer frequently accessed blocks. From the GFS master server a client obtains the information about which chunkserver to contact for a given file. The transfer of data is once more directly between the server holding the data and the requester. Usually, three replicas of any chunk are stored in the system for load balancing and data reliability reasons. In contrast with most other cluster file systems, fault-tolerance has been an important aspect during the design of GFS. As pointed out earlier, each chunk is replicated on several chunk servers. Depending on the number of replicas, which can be adjusted by the client, this approach makes data loss very unlikely. Since there is only a single instance of the master server for simplicity reasons, this server presents a single point of failure. Fast recovery of this server process using replicated operation logs or the restart of another new server process in the case of a system crash guarantees metadata access to the clients. Corruption of data by failing storage media is prevented by the use of checksums. Currently, Google deploys multiple GFS clusters, the largest with 1,000 storage nodes and over 300 terabytes of disk capacity, accessed by hundreds of clients.

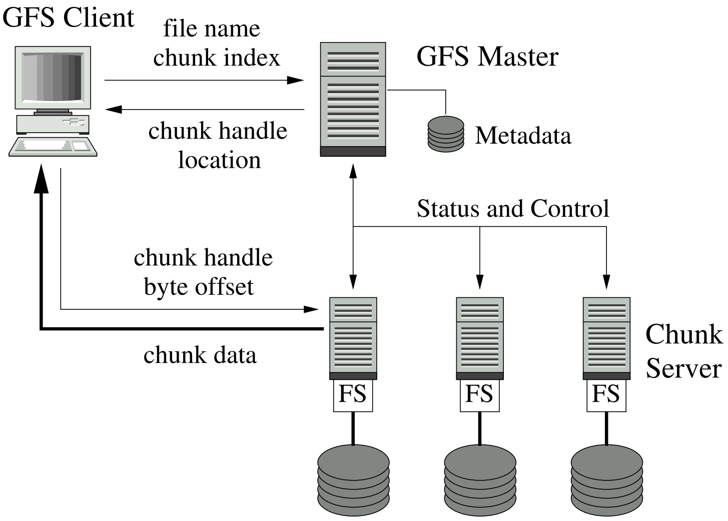

FIGURE 2.4.: Schematic overview of the Google File System architecture.

2.3. Integrated Mass Storage Systems

The systems discussed in this section are not intended or specifically designed to be used as secondary storage architectures for clusters. Unlike the distributed file systems discussed above or the block-level systems of the next section, they integrate secondary storage, usually disks, and tertiary storage, i.e. tapes, into a single consistent system. Here, a cluster and its storage capacity may only be one component among others. Therefore, the description of each system is kept short. None the less, such integrated architectures are commonly used, for instance in the context of high energy physics experiments, and, of course, they also represent an important architecture for distributed storage.

2.3.1. HPSS

The High Performance Storage System (HPSS) [85] is a hierarchical archival storage system, designed to manage petabytes of data. It is specifically aimed at moving large data objects between primary, secondary, and tertiary storage.

The architecture of HPSS roughly comprises the following constituent parts: the clients, the servers, a data network, a control network, and the actual storage devices, see figure 2.5. Clients are compute nodes, on which the need for data access arises. HPSS servers are responsible for establishing a transfer session between the clients and the actual storage devices using the control network. They can be classified according to their client interaction: some of the servers are contacted by clients in order to set up a data transfer, others are only contacted by other servers. The management operations directly initiated by clients include services for the location of the correct server (location server), the translation of a human-oriented name to an HPSS object identifier (name server), the file abstraction (bitfile server), or the interface to other systems (gateway server). Internal services take care of the placement of data on the storage devices (migration-purge server), the dereferencing of storage abstractions (storage server), or the management of physical volumes (physical volume

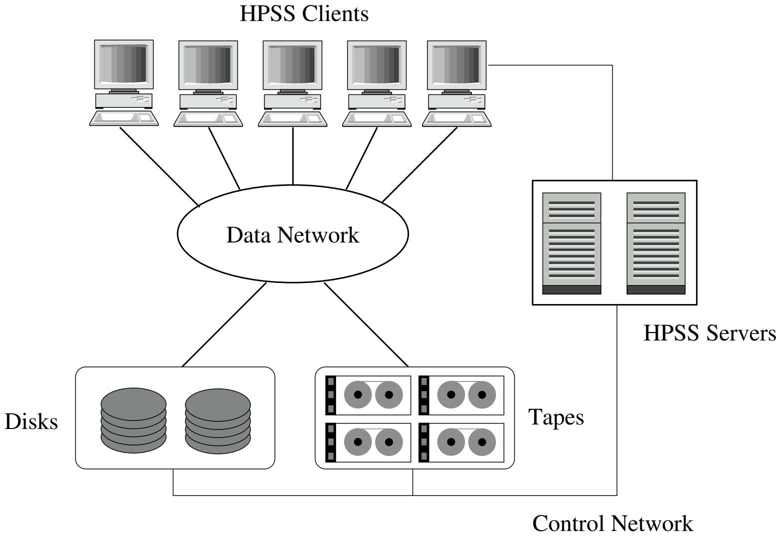

FIGURE 2.5.: Schematic overview of the network centered HPSS architecture. The HPSS servers manage the metadata information and establish the connections between clients and storage devices.

library and repository). A special class of HPSS servers are the so-called *movers*. A pair of mover tasks is actually transferring the data directly between storage and client using the data network. The other servers are not involved in this transfer.

Data caching inside HPSS is realized in terms of staging: frequently accessed data can be buffered on disks, instead of accessing a tape every time the file is requested. This feature can be controlled by clients or is done by HPSS automatically. Aside from this, there is no client-side caching in HPSS. Consistency of data is enforced by serialized data access. However, parallel access to data is provided by HPSS using collective requests and third-party-transfers: requests to a single file are merged into a single request sent by one compute node to the HPSS system. HPSS splits the request again and initiates the data transfers between the storage devices and the processes. Since the process sending out the request is neither the source nor the destination of the request, such transfers are referred to as third-party transfers.

HPSS offers several interfaces to clients for data access, for instance a native user library, NFS, FTP, or MPI-IO. The name space is unified for the whole system. Since the requests for data are traveling through a collection of servers, the latency of data requests is high. However, since HPSS is designed for transfers of very large data objects, latency is not a major concern.

2.3.2. CASTOR

The CERN Advanced STORage manager (CASTOR) [86, 87] is a hierarchical storage management system designed for the storage of data from high energy physics experiments. Like HPSS it integrates disk and tape storage into a single storage system with a consistent name space, see figure 2.6. The client interface to the CASTOR system is the Remote File I/O library (RFIO). This library is the interface to the central component of the system: the RFIO daemon (RFIOD). Clients always access

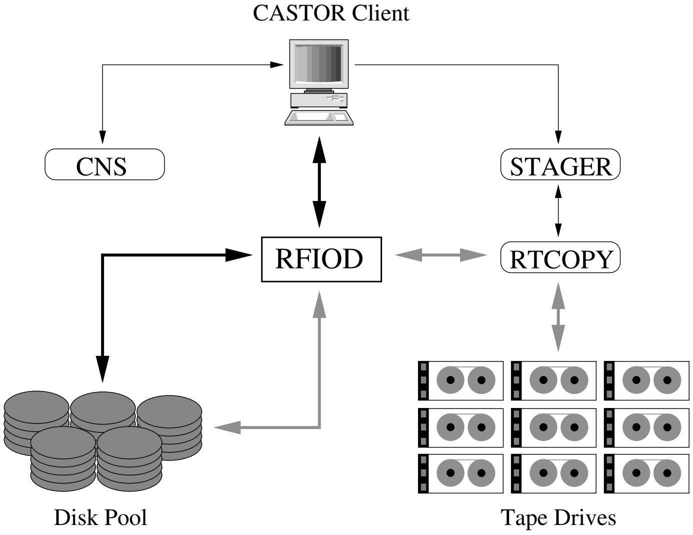

FIGURE 2.6.: Schematic overview of the CASTOR architecture. Thick black arrows indicate read and write requests by clients, grey arrows show the path of data that is staged from tape to disk or vice versa. Thin arrows show some of the control data paths.

data resident on disks, no direct access to tape is granted. The access is managed by the RFIOD. In order to open a file, the client contacts the CASTOR Name Server (CNS) to get the CASTOR file name. If the file is already staged from tape to disk, it is accessed by means of the RFIOD. If the file is only on tape, the client requests the stager to initiate the data transfer from tape to disk. The RTCOPY server is the CASTOR tape mover. For the RFIOD the RTCOPY is simply another client accessing the disk pool.[4] The stager, the RFIOD, or the RTCOPY server can be set up in a distributed fashion in order to increase performance and failure resilience. In CERN's installation with more than 2 petabytes of storage and over 9 million files, all servers additionally deploy fault-tolerant hardware, such as duplicated system disks or redundant power supplies.

2.3.3. Storage Tank

The Storage Tank system [88], depicted schematically in figure 2.7, is a storage management solution aimed at heterogenous environments, i.e. clients with different operating systems. Just as the other two systems presented in this section, it provides a unified consistent name space. A storage area network is used for the interconnection of clients and storage devices. The servers either use a dedicated private network or they can also be connected to the SAN used by the clients to transfer data. A crucial difference, however, is the caching policy. Any Storage Tank client must be installed with IFS, the Installable File System.[5] IFS redirects all metadata operations to the Storage Tank servers, which are the only instances in the system performing metadata changes. In order to provide low latency data access, data, locks, and metadata information are aggressively cached on the client

---

[4] The description given is reduced to the general functionality. The management of the tapes involves several additional servers, such as the Volume Manager and the Queue Manager.

[5] There exists also a native API for special applications, such as database management systems, to bypass the file system layer.

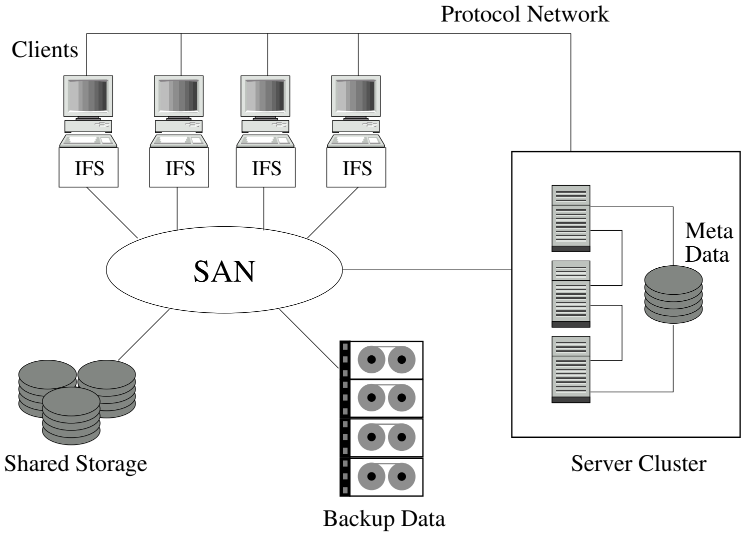

FIGURE 2.7.: Schematic view of the Storage Tank architecture. Clients access data by means of the Installable File System (IFS). A storage area network connects client, storage devices, and the metadata server cluster.

side by IFS, eliminating the servers as intermediaries in the case of repeated access to the same information. IFS provides exact local file system semantics by using distributed data locks for cache consistency and file locks for synchronization of concurrent access.

Storage virtualization is realized by so-called storage pools. A storage pool is a compound of multiple storage devices, which appear to the application as a single large storage space. These pools can be set up flexibly, according to the users' needs. For instance, pools can differ in terms of reliability, access latency, or bandwidth.

During normal operation the load on the servers is automatically balanced. In the case of a server failure the load is redistributed to the remaining ones. In general, media failures must be handled by the devices themselves. However, Storage Tank provides backup for device failures and supports RAID and mirroring devices to increase data reliability and availability.

2.4. Distributed Block Level Systems

Providing access to storage on block device level, rather than on file system level or by means of a user library, is the most generic approach. The block device interface is well defined and the management of blocks is assigned to the next higher abstraction layer. Usually, this upper layer corresponds to a file system, but in some cases the application itself is in charge of block handling. For instance, some database management systems prefer to act directly on block devices and bypass file system caches in order to improve their performance [89, 90].

Distributed block devices as discussed in this section come in two flavors. Firstly, point-to-point devices, such as NBD, ENBD, or DRBD, that masquerade remote hard disks, partitions, or files as local devices. These systems are intended for access by one client only. Any access to these devices is mapped to requests for the corresponding remote resource. Secondly, shared devices, such as RAID-x OSM, Petal, or DRAID, allow for concurrent access to shared resources.

As the ClusterRAID also offers access to storage on block device level, these systems are discussed in the following paragraphs with respect to their architecture, features, and drawbacks.

2.4.1. NBD and ENBD

Network Block Device is the general term for a software system that is able to masquerade remote resources (files, partitions, or disks) as local devices. Any local access to a network block device is transferred to a remote host, where the request is served, and the result is sent back over the network.[6] Network block devices differ from classic networked file access solutions such as NFS or AFS [91] in that access is at a lower level than the file system layer. This allows any file system structure to be supported on the device once it has been set up. The latency imposed by a network layer in the data path is negligible compared to the latency contributed by the hard disks whose seek time and rotational delay is of the order of several milliseconds. Standard network technologies such as Ethernet have data transmission latencies of up to several hundred microseconds at the most. Thus, the expected latency increase is about 10% or less. The same holds true for bandwidth restrictions. If a gigabit type network technology is used, the hard disk is the limiting component. As current trends indicate, the throughput increase in network technologies will be much faster than that of hard disks. Network block devices thus provide an elegant and reasonably efficient way to access remote resources. As the architecture of the ClusterRAID is based on the abstraction of network block devices, the description of this type of distributed block level system will be more detailed than for the other systems.

The very first implementation of a network block device for Linux, named *NBD*, was started in March 1997 [92]. It has become a part of the standard Linux kernel with version 2.1.67 and is still maintained in current kernel releases. NBD consists of three components: on the client-side the NBD block device driver and the NBD client, and on the server-side the NBD server daemon. The server node exports a local resource and waits for clients to connect to its listening TCP/IP socket. Once a connection is successfully established, the connection is handed down from the client to the driver. The client daemon exits, and the kernel driver on the client directly communicates with the user space NBD server on the server side. However, this asymmetric connection introduces one of NBD's main drawbacks: responsiveness. In order to serve user requests, the client system is very often in kernel mode, either to hand requests to the driver or to handle the transfers over the network. Since kernel code is not descheduled in Linux kernels of the 2.4 series (this changed with the 2.6 series), other processes can and will get only very limited processing time. Another drawback of NBD is its limited number of features: aside from its basic functionality it offers almost no extra features, for instance for fault-tolerance, performance optimization, security, or behaviour configuration. However, NBD is a reliable, stable, and easy to set up software system that provides good performance, as will be shown in chapter 6. The NBD has been used extensively for various test rigs of the ClusterRAID prototype.

As its name indicates, the *Enhanced Network Block Device* (ENBD) [93] tries to overcome the lim-

---

[6] This is true for any access that reaches the network block device driver level. If the operating system can serve a request from local caches, the driver is never asked for data and, thus, no network transaction will be initiated.

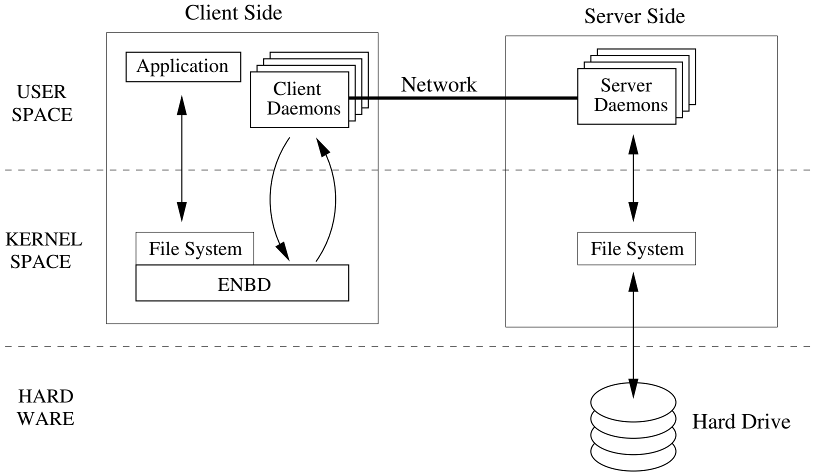

FIGURE 2.8.: Schematic overview of ENBD.

itations mentioned in the former paragraph. Since it is based on NBD, the architectures of both systems are very similar. Figure 2.8 shows a schematic view of the three active ENBD components: driver, client, and server. As for NBD, the driver provides the block device abstraction, while client and server establish the network connections. A major difference is the fact that the driver does not directly transfer requests over the network. After the connection has been established, the socket is not handed down to the driver, but kept in the client daemon instead. The client daemon is responsible for exchanging requests with the server. Requests for ENBD are picked up from the driver by the client daemon and thus the whole networking can be done in user space. This has several advantages. It overcomes the responsiveness problems of NBD, since the time to get a new request from the driver is short compared to the time needed to send it over the network and receive the reply. Furthermore, user space networking allows for an easier adaptation of ENBD to other network technologies as it allows for easier implementation, debugging, and testing.

In addition to the basic architecture and protocol of NBD, the Enhanced NBD provides failure-mode and redundancy features, such as automatic reconnect and reauthentication after temporary network failures. It can run over several communication channels at once, possibly through different routes, which adds redundancy. The channels are demand driven, so a dead channel results in an automatic fail-over to the remaining live channels. The multiple connections maintained by ENBD work asynchronously and this gives rise to a pipelining effect that speeds up the protocol, especially on platforms with multiple CPUs. A pair of auxiliary daemons, one on the server and one on the client side, help establish connectivity after reboots by maintaining persistent information about the desired state of the connections and initiating new connections when necessary. Consistency across system crashes, at least of journalling file systems set up on top of ENBD, is guaranteed as ENBD maintains write ordering. Coherence on the client side is guaranteed by the kernel, which serves reads on the device from its own cached image of the changes it has made. ENBD itself has no size limitations apart from those imposed at the client end by the kernel on any block device and by the file system layer at the server end.

The intended use of ENBD is among trustable nodes inside a cluster. However, ENBD does allow data to be encrypted using the Secure Socket Layer (SSL) [94], a security protocol that provides

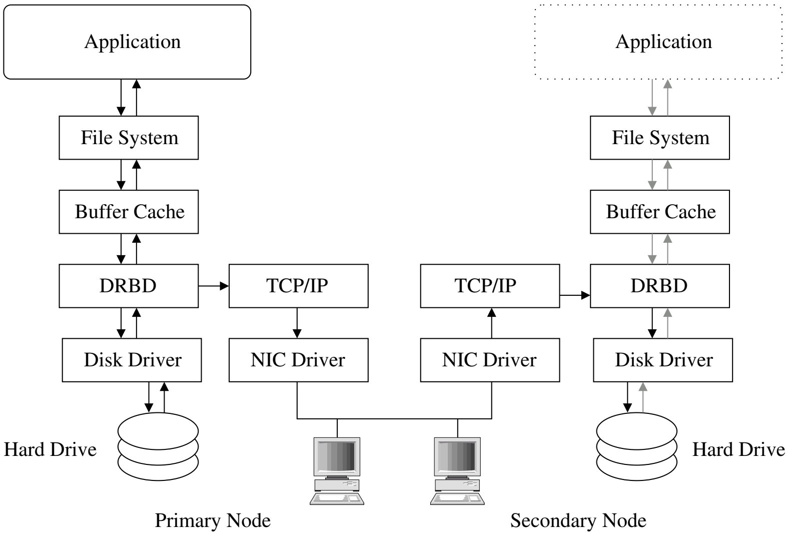

FIGURE 2.9.: Schematic overview of DRBD. The DRBD layer forwards write requests to both the local device and the remote device. Read requests are always served from the local device.

communication privacy. In addition, server and client exchange a signature when establishing the first connection, which is used in all later reconnection attempts in order to authenticate the other side.

In order to improve write performance and to avoid unnecessary network transactions, ENBD is able to check if the transfer of data can be omitted by comparing checksums of both the remote data on the resource and the data to be written.

This feature list, especially the built-in redundancy features, make ENBD the clear choice to serve as the mechanism to transport blocks between different nodes inside the ClusterRAID system.

2.4.2. DRBD

The *Distributed Replicated Block Device* (DRBD) [95] is a Linux kernel module for building two node high-availability storage clusters. The two nodes of such a cluster have mirrored copies of the stored data. Reads are served from the local devices, while writes are also mirrored to the other node. Logically, DRBD is a control layer upon the real block devices, see figure 2.9. All requests are intercepted by the driver before they are forwarded to the local device and – in the case of write requests – also to the remote device. Each of the two nodes has assigned a certain state or role, *primary* or *secondary*. For consistency reasons, write access to the DRBD device is only granted to applications if the node is in primary state.[7]

The main feature of DRBD is the ability to configure the coupling between the nodes. This coupling has influence on both the performance and the data reliability. Three protocols, referred to as *A*, *B*, and *C*, define the degree of coupling, which is mainly characterized by the point in time a write request is signaled as completed to the upper layers. Protocol A signals the completion of a write operation already after the request has been completed on the local device. Protocol B acknowledges the request as soon as the remote node has received the request and the local request is finished. Protocol C is completely synchronous. Here, a write request is regarded as complete as soon as the request is written to both disks, the local and the remote. Protocols A and B impose the problem of write ordering: a request may be signaled as finished to the upper layer, for instance a file system, before it has reached the remote hard drive. Since journalling file systems or databases rely on the order of writes for their logs, a failure of the primary node may leave the secondary node, from which the system must be recovered, in an inconsistent state. In order to avoid these problems DRBD tries to recognize and preserve any data dependencies. This is done by introducing so-called *epoch sets*, lists of recently issued requests. If a new request is received by the driver after an older request has been acknowledged, the completion of all requests up to the new one is enforced by write barriers on the remote node. In this way, all pending requests are finished before any new ones are processed and no possible write-after-write hazard arises. Since it is completely synchronous, this problem does not occur with Protocol C.

---

[7] This holds true for the current 0.6 version of DRBD, but is announced to be changed in the next 0.7 release.

In order to reduce the network traffic and to speed up the resynchronization after a node failure, DRBD maintains a bitmap with modified blocks. Upon return of the failed node, only the blocks marked modified since the degradation are synchronized. In addition, it is planned to add a checksumming mechanism similar to the one of ENBD described in the former section, in order to reduce data transfers over the network in case that a full resynchronization of a node is necessary.

2.4.3. RAID-X OSM

RAID-X OSM [96] is a distributed disk array architecture for clusters. The special characteristic of this system is the way in which the data is distributed across the integrated disks. Although reliability is achieved by mirroring, the distribution of the stripes is more sophisticated than for the other RAID levels. Figure 2.10 illustrates the scheme. Data is striped across several nodes in the cluster. However, the mirrored stripes are written to a single additional node only. This distribution scheme is referred to as *Orthogonal Striping and Mirroring (OSM)*. The main feature of this orthogonal mapping of data and image blocks is an increase in write performance compared to other distributed RAID systems. Instead of synchronizing accesses to multiple disks, the mirrored data blocks can be written in streaming mode to a single device.

RAID-X provides a single I/O space (SIOS) [97, 98] for all nodes in a cluster, the total capacity of all disks is usable at any node. Such a distributed system of course requires a distributed locking scheme in order to provide data consistency. RAID-X OSM uses multiple-read and single-write locks for all I/O operations on the cluster nodes. These locks are implemented in so-called *Cooperative Disk Drivers (CDDs)*. All CDDs in the system synchronize their access to data blocks. Similar to the device masquerading of ENBD, these drivers also allow to access a remote disk over a network connection. In contrast with ENBD, all networking is done at kernel level.

The RAID-X OSM system, however, has one crucial drawback: its scalability is limited by both the distributed locking and the block distribution scheme. Since for any access the corresponding lock must be requested, the probability of concurrent locking requests increases with the number of nodes. This congestion limits the overall throughput, especially for write accesses. The distribution scheme presents an even more severe problem. In addition to its non-optimal space overhead of 50%,

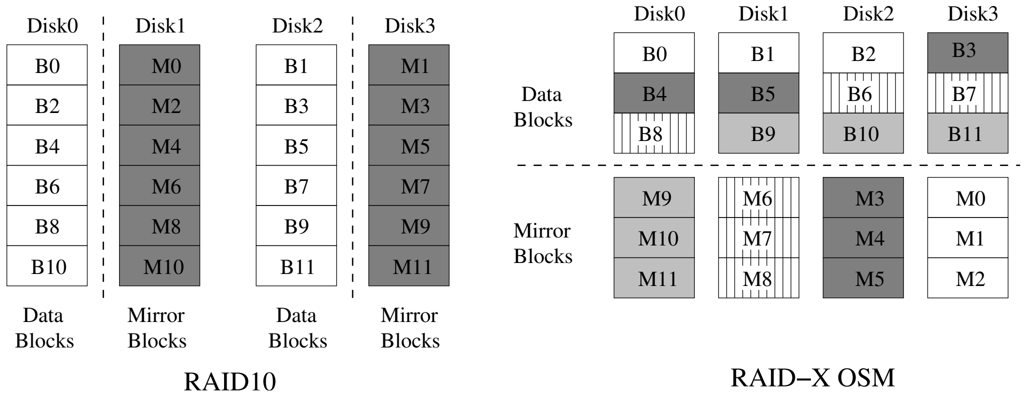
Disk0
Disk1
Disk2
Disk3
B0
B2
B4
B6
B8
B10
M0
M2
M4
M6
M8
M10
B1
B3
B5
B7
B9
B11
M1
M3
M5
M7
M9
M11
Data Blocks
Mirror Blocks
Data Blocks
Mirror Blocks
RAID10
Disk0
Disk1
Disk2
Disk3
Data Blocks
Mirror Blocks
B0 B1 B2 B3
B4 B5 B6 B7
B8 B9 B10 B11
M9 M6 M3 M0
M10 M7 M4 M1
M11 M8 M5 M2
RAID–X OSM

FIGURE 2.10.: Comparison of the block distribution for RAID-10 and RAID-X OSM. Although the space overhead is the same as for RAID-10, the reliability of RAID-X OSM is worse due to fact that the loss of a device means a loss of both data and redundancy blocks. The loss of two devices means a loss of data for RAID-X OSM, while the RAID-10 block arrangement gets over certain combinations of a double device failures.

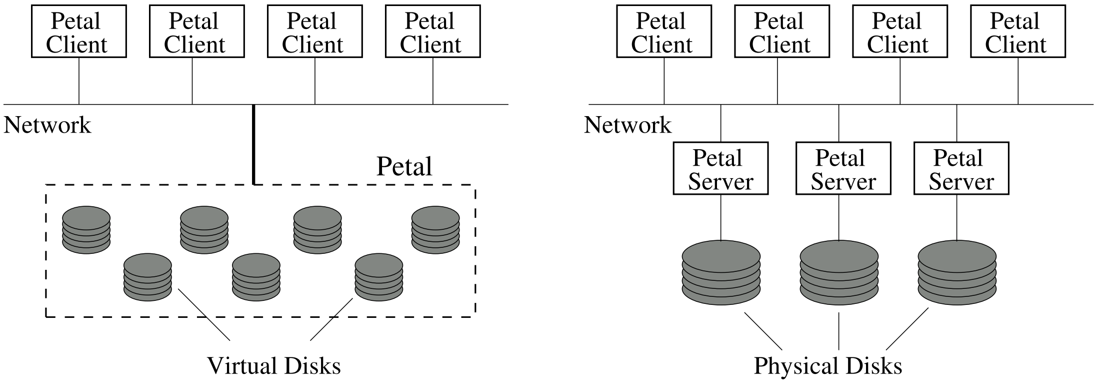
Petal Client
Petal Client
Petal Client
Petal Client
Network
Petal
Virtual Disks
Petal Client
Petal Client
Petal Client
Petal Client
Network
Petal Server
Petal Server
Petal Server
Physical Disks

FIGURE 2.11.: Logical (left) and physical (right) view of the Petal system.

orthogonal striping and mirroring does not protect sufficiently against data loss for larger setups. The loss of any two nodes will result in the loss of data. As will be shown later, the mean time to data loss is inversely proportional to the number of nodes. Therefore, this system cannot be scaled to hundreds or even tens of nodes.

2.4.4. Petal

Petal [99] is a distributed block level storage system exploiting the abstraction of *virtual disks* in order to hide the distributed nature of the system. These virtual disks provide the same interface as local disks. However, the virtualization of physical disks allows Petal clients to share both the capacity and the performance of the underlying hardware. Furthermore, it allows for both a uniform load and capacity balancing. Figure 2.11 illustrates the logical and physical views of the Petal system. The Petal servers use distributed consensus algorithms to communicate the global state of the system.

One of the major strengths of Petal is the consequent integration of fault-tolerance mechanisms. It is one of the few systems that can handle disk, node, and network failures. To protect against data loss it uses the chained-declustering [100] scheme. This replica-based algorithm allows a Petal system to be installed over geographically distributed sites. Petal is transparently reconfigurable and can be incrementally expanded, for instance by adding new servers or disks.

All virtual disks are visible to all Petal clients, but there is no protection of a client's data from accesses by other clients. This control is left for higher level services, such as the Frangipani distributed file system [101], which has been built for use on top of Petal. Frangipani provides a coherent, shared access to the same set of files and profits from the features of Petal, such as its fault-tolerance or extensibility.

2.4.5. Tertiary Disk

The Tertiary Disk [102] storage system of the University of California in Berkeley is one of the first and largest (in terms of disks used in the system) projects to build a storage system with commodity hardware. Its primary design goal was to build a cheap and reliable storage system without relying on special hardware, but rather on standard PCs and disks. Together with the DRAID system, which will be discussed in the next section, it thus comes closest in spirit to the ClusterRAID project. Unlike most other systems discussed in this section, Tertiary Disk achieves the necessary reliability by combining software and hardware mechanisms in order to tolerate component failures.

A *node* in the Tertiary Disk system consists of two PCs which share disks. Each of the PCs has four SCSI controllers, which are connected to a disk enclosure hosting the shared disks. The number of disks per SCSI string varies between 8 and 14. The double-ended SCSI string ensures data access even in the case of a failed node. During normal operation each of the PCs accesses only half of the attached disks. Protection against data loss by disk failure is achieved by establishing stripe and parity groups over the nodes in the system. The data distribution is organized such that a SCSI string always contributes only one disk to one of these stripe groups. Thus, no single failure can lead to data loss. The Tertiary Disk system is the basis for (and intended to be used with) the xFS [103] distributed file system.

2.4.6. DRAID

DRAID [104] is a distributed, fault-tolerant block device architecture for PC clusters. One of the main design goals of DRAID – as for RAID-X OSM – was to implement a single I/O space in order to provide a better suitability of the system to applications requiring parallel I/O. In contrast with RAID-X OSM, DRAID does not rely on mirroring or parity schemes. Instead, Reed-Solomon codes are deployed, which allow the toleration of a site-configurable number of failures. As a special variant of these codes is also used in the ClusterRAID prototype implementation, they will be discussed in detail in chapter 4.

DRAID nodes are arranged in a two-dimensional fashion, the so-called *DRAID matrix*. Each row of this matrix contains N nodes for data storage and κ nodes for redundancy information. A logical block of the virtual DRAID device consists of N data segments and κ redundancy segments. On the basis of these $N+\kappa$ segments the logical blocks are distributed over the nodes in a matrix row. This approach allows for an easy expansion of a DRAID system: new rows can be added to the matrix without recomputing the redundancy information. However, the optimal space efficiency that is usually inherent to the Reed-Solomon codes is lost to a certain degree, since the additional redundancy devices of new rows do not increase the number of tolerable failures for the whole system.

If a block is written to the DRAID device, it is striped over the N data nodes in a row of the matrix. The result of the Reed-Solomon encoding applied on these data stripes is stored on the κ redundancy nodes. As the data is stored in a redundant fashion, it is sufficient if only N of the $N+\kappa$ segments of the block are available to a requesting node in the case of a read. It is advantageous, however, if the segments of the N data nodes are used, because this avoids the computing intensive decoding of the redundancy information.

The DRAID prototype is implemented as a set of kernel modules for the GNU/Linux operating system in a completely symmetric peer-to-peer architecture. The protocol used for the communication between the nodes is UDP, deployed at kernel level. While DRAID's write performance can saturate a Gigabit Ethernet link, its read performance suffers from the missing congestion control in UDP and in the communication layer of DRAID. The responses to a read request and to the requests issued due to the implemented read-ahead congest the corresponding switch port, leading to packet loss and a performance drop. However, the authors plan to investigate the use of other protocols and the implementation of specialized scheduling provisions to overcome this limitation.

It is important to note that of all the distributed systems presented so far, DRAID is the only one that is able to tolerate more than just one disk failure. As the reliability discussion in section 3.5 will show, this capability is mandatory when building systems for larger clusters consisting of commodity components.

2.5. Distributed RAID

Network block devices have been developed with the aim of expanding the concept of a local RAID to systems that include remote devices as their constituent parts. The block device interface allows for a transparent exchange of directly attached devices with remote ones, and thus enables a local RAID over remote disks.

As a first step towards a reliable mass storage system constructed from commodity components, a prototype of such a Distributed RAID system has been set up using existing building blocks [105]. Due to its large number of features and its excellent support, the ENBD was the network block device of choice. Moreover, ENBD's implementation of networking in user space allowed for a relatively easy port to other network technologies. The adaptation of the system to the SCI interconnect demonstrated the principal independence of the system from the underlying network, and helped to reduce the CPU load imposed by the network traffic. For the RAID layer the standard module as included in the Linux kernel has been used.

Figure 2.12 shows the architecture of the Distributed RAID system. A number of backend servers export their local resources by means of network block devices to a frontend server. A frontend server can span a local RAID over the imported devices. The RAID device can then be served to client nodes via a network file system, just as the server would do with a RAID over local devices. Optionally, a spare frontend server can take over services by connecting to the backends in case of failure of the primary frontend server.

This distributed architecture of course rivals installations of standard file servers with local devices,

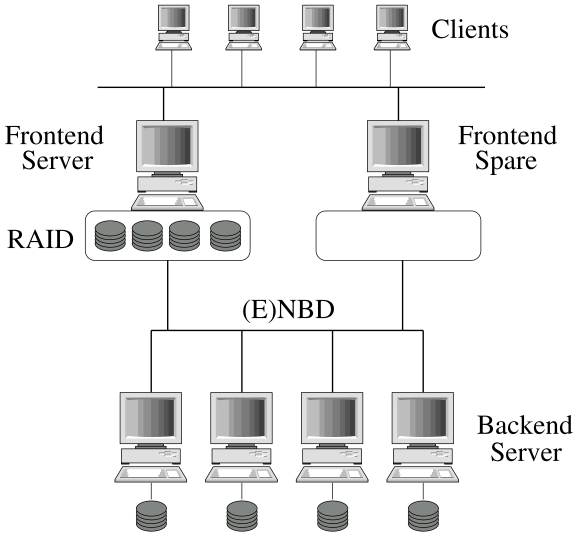

FIGURE 2.12.: Sample architecture of a Distributed RAID. The backend servers export their local resource via a network block device to a frontend server. Since the remote disks are masqueraded as local devices, the frontend node can deploy a local RAID over remote disks.

such as depicted in figure 2.1. Therefore, the following discussion of the advantages and drawbacks of the Distributed RAID approach will include a comparison with the standard file server solution, where relevant.

A first architectural feature of the Distributed RAID is the ability to consolidate (possibly unused) disk space from several nodes into a single large storage device. The device uses the same RAID mechanisms that a standard file server applies, and thus is able to protect from data loss by single disk failures. Furthermore, since the backends' resources appear as usual block devices on the frontend, the failure or unavailability of a complete backend node (for instance due to network interconnect or power supply failure) results solely in one RAID component marked as faulty on the frontend. All data is still accessible and available to clients. As already mentioned, a faulty frontend node can be transparently replaced by a failover server.[8] Thus, a Distributed RAID has no single point of failure, since it can tolerate any disk, net, or node failure without loosing access to data. A network interconnect or power supply failure in a standard file server will make the data at least temporarily unavailable.

Furthermore, a "device" failure in the Distributed RAID does not necessarily mean that the data on that device is lost. A temporary network interruption may be misinterpreted as a backend failure. The device in the backend and the data stored on it remains intact, but unavailable. ENBD's features for a fast resynchronization mentioned in section 2.4.1 and activities such as FastRAID [108] are designed to adapt the standard RAID to installations, in which network block devices are part of the array.

The modularity of the Distributed RAID approach allows for easy exchange of building blocks. The network block device component in the system could be replaced by any system offering a block

---

[8] This is a standard approach for NFS servers connected to external disk arrays. The Heartbeat [106] package of the Linux High Availability Project [107] project is a widespread implementation to increase data availability in this way.

	Random Seeks [1/s]	Avg. Seek Time [ms]	Latency Increase [%]
Hard Disk	200	5	-
ENBD on FE	175	5.7	14
ENBD on GbE	179	5.6	12
ENBD on SCI	185	5.4	8

TABLE 2.2.: Bonnie++ seek time results on a point-to-point ENBD. The latency increase is calculated relative to the seek time of the hard disk.

device interface to a remote resource, such as iSCSI. In addition, the RAID level can be chosen according to the application level requirements, but the RAID module could also be replaced by other systems, such as the LVM software, to administer the multiple disks.

The main architectural difference to a standard RAID is the introduction of a network in the data path between application and disk. Therefore, the impact of this additional component on the performance needed to be evaluated. As already mentioned, the increase of data access latency by the additional network in the data path should be of the order of 10%. Measurements with the bonnie++ I/O benchmark [109] confirmed these expectations, as shown in table 2.2.

Current disks cannot saturate gigabit networks, such as Gigabit Ethernet or SCI. A comparison of the compound growth rate of disk throughput and new developments in network and bus technologies, such as InfiniBand, 10 Gigabit Ethernet or PCI-X, indicate that this gap will widen in the near future. Therefore, the network should not prove to be a throughput limitation for the Distributed RAID.

In order to verify this assumption, some very simple tests of ENBD as the central building block have been performed. Figure 2.13 shows ENBD's read performance for different networks. If the network is faster than the underlying resource, the ENBD client can access it at nominal speed. Figure 2.14 shows oscillations in the ENBD write throughput. At regular intervals the network does not transfer any packets. This is due to the buffer management of the virtual memory subsystem. Local buffers are filled, before data is flushed to the underlying device. Once the available buffer space is exhausted, backpressure prevents the client from sending more data over the net. These oscillations can be controlled by adjusting the buffer flush parameters of the server's virtual memory subsystem.

The behaviour of a Distributed RAID system in case of a backend failure is shown in figure 2.15. After about 30 seconds of operation a network cable was unplugged to simulate a backend failure. The RAID device could not get any more data from one of its clients, and the network transfer rate dropped to zero. Once the ENBD driver has reported the device as being faulty to the RAID, the latter starts to serve data from the remaining servers after about 60 seconds. The data input rate on the frontend node is slightly reduced due to the fact that one of the servers is not available anymore and its data must be reconstructed from parity information. After the network cable has been plugged in again, the ENBD components automatically reconnected. The removal of the device from the RAID and its reintegration triggered the background resynchronization, which is not shown in the figure.

Obviously, since a Distributed RAID is just a local RAID over remote disks, it inherits many of the characteristics and problems of a local RAID. Among these are the space overhead, the small-writes-problem [110], and the limited scalability. Since standard RAIDs can only tolerate the loss of a single component, the number of devices in an array is limited by the probability of a second failure

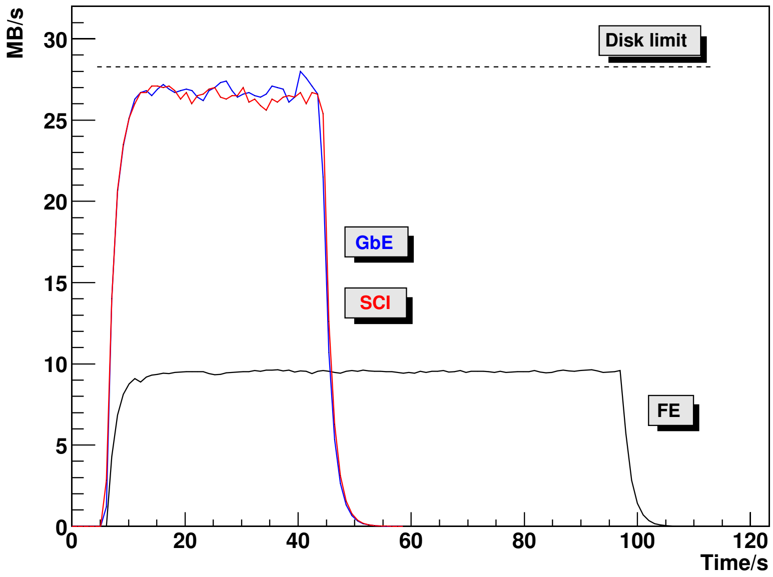

FIGURE 2.13.: ENBD read throughput for different network technologies. If Fast Ethernet (FE) is used, the network is the limiting factor. For faster networks, such as Gigabit Ethernet (GbE) or SCI, the bandwidth of the underlying disk is limiting the overall throughput.

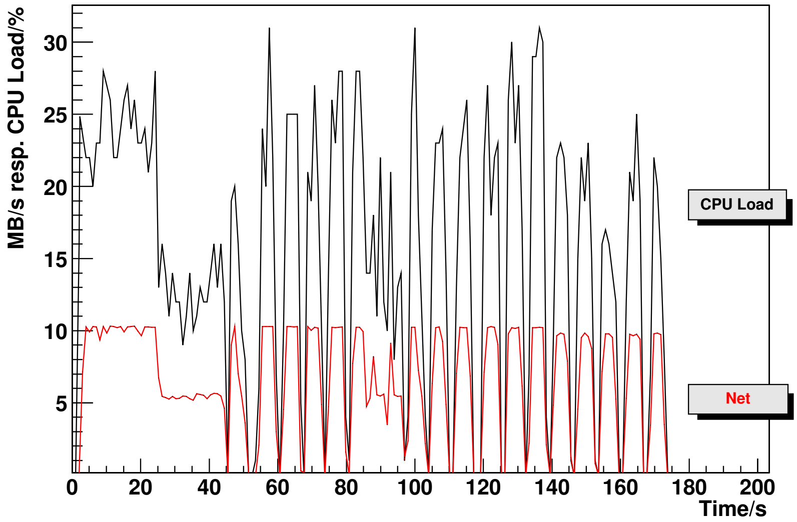

FIGURE 2.14.: ENBD write throughput and CPU load vs. time for a write transfer of 1 gigabyte. The backpressure of the server-side virtual memory subsystem's buffer flush strategy induces an oscillating transfer rate.

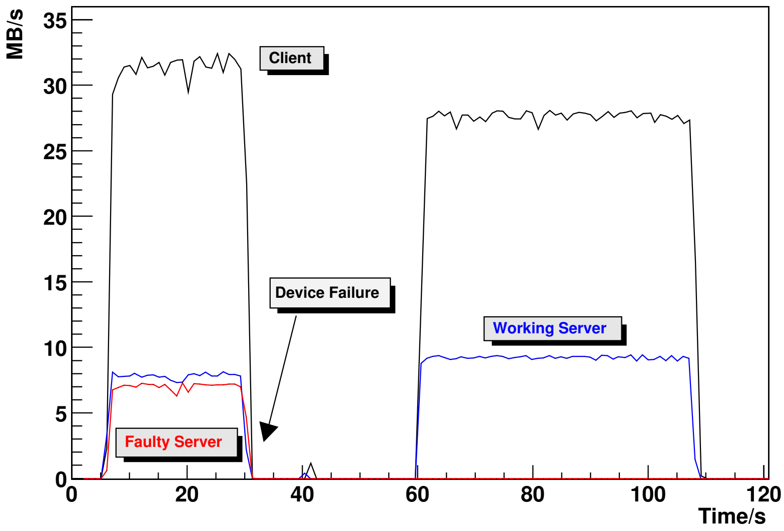

FIGURE 2.15.: Network traffic during a backend failure. The test setup consisted of four backends and one frontend node. Note: For reasons of clarity only two server graphs are shown, one of a working server and the one of the faulty server. The faulty server's curve is scaled by a factor 0.9.

while the first failure is not yet repaired. Hence, the number of backend nodes in a Distributed RAID is limited.[9] An additional problem of the Distributed RAID is the CPU load imposed by the network traffic. Each data access involves a network transaction and thus increases the load on the CPUs of both frontend and backend nodes. Although this problem can be alleviated by using the traffic reducing features of ENBD or low overhead networks, the CPU load remains a limiting factor, in particular during reconstruction.

Despite these drawbacks, a Distributed RAID offers the possibility to build a reliable mass storage system with commodity components. Details about the deployment of a production quality server can be found in Appendix A.2.

2.6. The need for something different

As already indicated in the introduction to this chapter, the requirements for mass storage are multifaceted, and so are the approaches to meet them. A summarizing comparison of all the architectures is difficult and maybe inappropriate, since they differ in many aspects, such as interfaces, hardware requirements, abstraction levels, reliability, and performance (in terms of latency or throughput). However, a characteristic common to almost all these systems is that they do not take into account the possible benefit of adapting the architecture to the characteristics of a broad range of applications, i.e. to their access behaviour and their inherent independence of tasks. In many applications read accesses tend to occur much more frequently than write accesses. Most data is written once but read multiple times. This applies, for instance, to all kinds of applications performing searches, data analyses, data mining, data retrieval, and information extraction. In addition, many applications can be split into several independent tasks processing independent data. Two examples of such applications that even combine these two characteristics, have been briefly examined in the introduction of this thesis.

---

[9] New developments such as the RAID-6 driver introduced in the 2.6 series of the Linux kernel are aimed at overcoming this limitation.

The MOSIX file system or the Google file system are examples of distributed storage systems that make use of their knowledge about the application. By moving the data to the application during system operation, MOSIX can even react dynamically to actual I/O needs. After processes' migration the data is accessible on the local node, reducing both the CPU load and the load on the network. The Google file system is intended to be used with one specific application, and hence the overall system was able to benefit from the co-design of the application and the mass storage system. For instance, the file system was simplified by relaxing the consistency model without a significant increase in the complexity of the API. In addition, application specific operations have been added to improve the system's performance.

Another drawback common to almost all mass storage systems is the neglect of sophisticated reliability mechanisms. The preferred approach to protect against data loss - if the system provides such a protection at all - is by generating replicas, which reduces the amount of usable storage capacity significantly. The Google File system, for instance, generates three replicas for any data chunk. Hence, only one third of the total capacity can be used for active data storage.[10] The ClusterRAID architecture as presented in the following tries to fill the gap and to provide a distributed reliable mass storage system for commodity off-the-shelf clusters with special consideration of the aforementioned aspects.

---

[10] It has to be noted, however, that GFS uses the replicas also for load balancing purposes.

3. ClusterRAID Architecture

Make the common case fast.
The RISC principle.

This chapter sets out design principles, architecture, and functional characteristics of the ClusterRAID system. This allows a discussion of the system's structure, functionality, performance, scalability, and reliability to be completely decoupled from a description of its concrete implementation.

3.1. Design Considerations

In the introductory chapter of this thesis, cluster applications from both science and industry were examined regarding their requirements for mass storage, their typical I/O behaviour, and their parallelism. The discussion showed that the total capacity needed in these applications is in the range of several hundred terabytes up to a few petabytes. All of them issue at least one order of magnitude more read requests than write requests and at least some of them are easily parallelizable, such as the data processing for high energy physics experiments. Although not all applications comply with these characteristics, in particular not with the need for a petabyte storage capacity or the trivial partitioning into independent tasks, a read-to-write ratio larger than one holds true for many applications. The design of the ClusterRAID architecture is led by a number of goals. However, inspired by the example applications, the most incisive goals are the optimization for the most frequent type of access, i.e. read requests, and the provision for the separation of an application into data-independent tasks.

Using large commodity off-the-shelf clusters as a computing platform introduces a problem any distributed system must necessarily take care of: the inherent unreliability of the individual cluster components. For a distributed mass storage system the protection against data loss given these frequent component failures is the main issue to be addressed here. Large installations will inevitably need to tolerate multiple, simultaneous failures [111]. Hence, the application of data protection mechanisms that can reconstruct data also in case of multiple failures must be considered. A comparison of different approaches to achieve data reliability has shown that erasure-resilient codes provide higher data reliability than replication-based schemes for comparable storage and bandwidth boundary conditions [112]. Therefore, the deployment of such codes is the method of choice to protect against data loss in the ClusterRAID.

Since the ClusterRAID provides parts of the operating system services, the amount of resources it consumes should be reduced to a minimum to limit the impact on user applications. Resources in this sense not only include CPU overhead and network bandwidth, but also the disk space overhead needed by the redundancy algorithm. The selected algorithm should require only a minimal space overhead, provide adjustable reliability, and have a reasonable computing complexity. Since the former two requirements may conflict with the latter, the algorithm should be amenable to optimization in software and hardware.

The application interface of the ClusterRAID should be the same as for a directly attached hard drive: the standard block device interface. This interface is well-defined and allows the ClusterRAID to be used as a direct replacement for standard directly-attached disks. It also enables the co-operation with other mass storage services that do not have their own built-in provisions for data reliability, such as MOPI or PVFS[1]. For these systems, the ClusterRAID can provide an underlying mass storage device that guarantees the required data reliability.

From an implementation's point of view the system should be developed in software rather than in hardware. Although a mainly software-based solution may induce more overhead, there certainly is a gain in flexibility and ease of development, considering debugging, for instance. However, the software should allow for a possible partial re-implementation of the most computing intensive parts in hardware. Since almost all cluster installations are based on Intel compatible CPUs using a GNU/Linux operating system, this should be a baseline environment the ClusterRAID software should run on.

To sum up, the ClusterRAID architecture and its prototype implementation as described in chapter 5 are in compliance with the following design goals:

- optimization for read accesses,
- optimization for data independent applications,
- adjustable reliability,
- reduction of network traffic to a minimum,
- reduction of disk space overhead to a minimum,
- minimization of CPU overhead,
- a block device interface,
- scalable for large setups, and
- runnable on the GNU/Linux operating system on Intel-based processors as a baseline.

3.2. Overview and Functionality

The key paradigm of the ClusterRAID architecture is to convert the local hard drive into a reliable device while preserving its application interface. A ClusterRAID device on a node is a virtual device, a proxy device of an underlying real hard drive. During normal operation, all requests are forwarded to this constituent device. All additional actions taken by the ClusterRAID system are transparent to applications. From their point of view, the ClusterRAID simply provides a reliable block device. If the local hard drive is operational, the ClusterRAID serves all *read requests* directly

---

[1] As already mentioned in section 2.2.4 there are activities, such as CEFT-PVFS, to add fault-tolerance functionalities to PVFS. With respect to the protection of data however, CEFT-PVFS uses a RAID-10 like approach, i.e. mirroring. This reduces the usable space by a factor of 2.

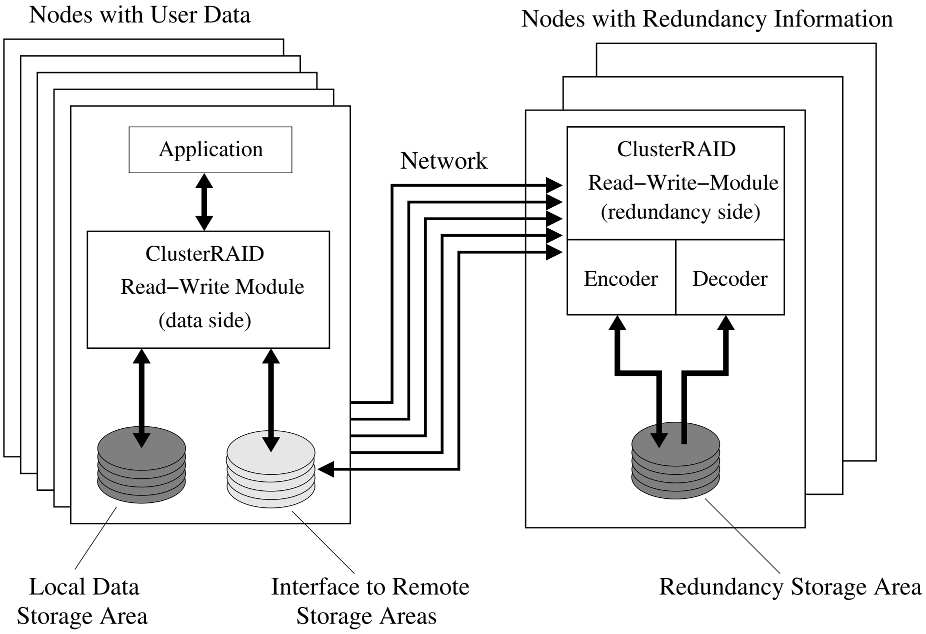
Nodes with User Data
Nodes with Redundancy Information
Application
ClusterRAID
Read−Write Module
(data side)
Network
ClusterRAID
Read−Write−Module
(redundancy side)
Encoder
Decoder
Local Data
Storage Area
Interface to Remote
Storage Areas
Redundancy Storage Area

FIGURE 3.1.: Functional top level view of the ClusterRAID system.

from its assigned underlying hard disk. No network transactions are required to do so. In addition to the redirection to the underlying device, *write requests* also initiate the generation of redundancy information, from which data can be reconstructed in case of a device failure. This redundancy information is transferred over the network and stored on remote nodes to protect against data loss by complete node failures. The fundamental approach of enhancing the reliability of a directly-attached device by remote redundancy information already covers several of the design goals exposed in the previous section. Since reads are processed locally, their performance in terms of throughput, overhead, or latency will not differ significantly from direct read accesses to a local device. No network transfers are necessary to serve them, relieving both the network and the host processor. The constituent device on a ClusterRAID node is not shared by multiple nodes for data storage, but is used exclusively by the local host. Network traffic to transfer the redundancy information to remote hosts on write accesses is inevitable and is common to all distributed systems. However, the amount of traffic depends on the redundancy approach chosen. As outlined in chapter 4, there are erasure-correcting codes that can reduce the space overhead, and thus the necessary network traffic, to the absolute theoretical minimum.

Figure 3.1 depicts a functional top level overview of the ClusterRAID system. The total storage capacity of all devices in the ClusterRAID is divided into storage areas for user data and storage areas for redundancy information. While data storage areas are assigned to the node they reside on for exclusive and asynchronous access, redundancy storage areas are used by remote nodes to store information needed to reconstruct data in case of a failure. Therefore, every data block in a local data storage area is assigned one or more redundancy blocks in the remote redundancy storage areas. A single redundancy block is usually shared by multiple data blocks or nodes, respectively. If a request fails, the data on the local device can be reconstructed using the redundancy information in the assigned redundancy storage areas and the data in the other data storage areas sharing the same redundancy blocks. The mapping of data blocks, the serving and supervision of requests, and the triggering of redundancy encoding and decoding can be encapsulated in a data-side ClusterRAID read-write module. On the redundancy-side, a corresponding read-write module is needed to store the redundancy information.

The redundancy information is generated by a redundancy codec. This codec also reconstructs requested data in the case of failures. For the update of redundancy information upon write accesses and for the reconstruction of data in the case of failures an interface to the storage areas of remote nodes is required. The actual encoding and decoding of data can happen anywhere in the data path, therefore the codec can reside on both the node with the data storage area or on the node with the redundancy storage area.

As indicated in figure 3.1, the total number of blocks in the redundancy storage areas will typically be smaller than the total number of blocks in the data storage areas in order to ensure a low space overhead for redundancy information. The number of redundancy blocks assigned to a data block in a data storage area – and thus the number of tolerable failures – will typically be larger than one. This ensures a significantly enhanced system reliability compared to the protection by a single redundancy block. This issue is discussed in detail in section 3.5 and expressed in equation 3.21, see page 66.

Although the functional depiction suggests that nodes in the ClusterRAID either store user data or redundancy information, the nodes can actually have both functionalities and store both block types. A set of data blocks and their assigned redundancy blocks is called a *redundancy ensemble*. The distribution of redundancy blocks of an ensemble over the nodes and devices in a ClusterRAID system is – aside from one constraint – arbitrary. In order to avoid that a single device or node failure results in multiple unavailable blocks in a single redundancy ensemble, a node must not hold more than one block of a specific redundancy ensemble.

The mapping between individual blocks in data storage and redundancy storage areas has to be performed in three scenarios:

- Writing to a data block requires to identify the corresponding redundancy nodes and blocks in order to store the redundancy information.
- Reconstructing data in the case of a data device failure requires to read the remote redundancy information and the data of the other data blocks sharing the same redundancy blocks.
- In case a redundancy device fails, a node holding a redundancy storage area must be able to contact the nodes holding data blocks that have assigned redundancy blocks on the failed device in order to regenerate the lost redundancy information.

A block in the ClusterRAID can be unambiguously identified by a tuple $\{n, b\}$, where n denotes the unique node or device ID and b denotes the block offset on the device. These tuples are used as input to the mapping relations. In addition, the letters N and κ are used for the total number of nodes or devices in the system and the number of redundancy blocks in one redundancy ensemble, respectively. This notation will be used throughout the whole chapter.

Figures 3.2 and 3.3 illustrate two simple examples for mapping relations between data and redundancy blocks. In the first distribution two devices – and the nodes they reside on – are dedicated to

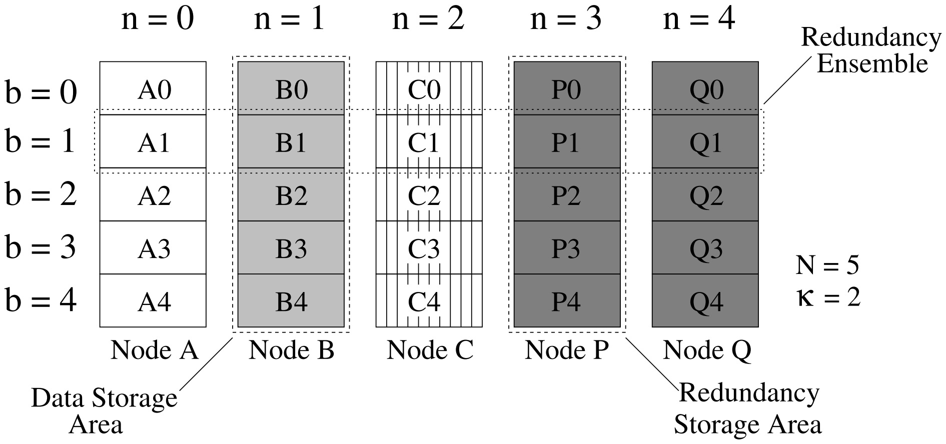

FIGURE 3.2.: Data distribution scheme with dedicated redundancy nodes. The nodes A, B, C only store user data, while the nodes P and Q only store redundancy information. Blocks on the same offset belong to one redundancy ensemble.

store redundancy information only ($n = 3$ and $n = 4$). The mapping relations for the three cases can be easily expressed by the following relations:
A data block in a data storage area, i.e. $0 \leq n < N - \kappa$, is mapped to κ redundancy blocks:

$$\{n,b\} \mapsto \bigcup_{i=0..\kappa-1} \{N - \kappa + i, b\} \tag{3.1}$$

For reconstruction, the remaining data blocks in the redundancy ensembles are required in addition to the blocks holding the redundancy information:

$$\{n,b\} \mapsto \bigcup_{i=0..N-\kappa}^{i \neq n} \{i,b\} \tag{3.2}$$

Finally, the data blocks associated with a redundancy block, i.e. $N - \kappa \leq n < N$ can be calculated by

$$\{n,b\} \mapsto \bigcup_{i=0..N-\kappa} \{i,b\} \tag{3.3}$$

This distribution with dedicated redundancy nodes is comparable to the RAID-4 scheme discussed in section 2.1: every data block on a data node has assigned the blocks at the same block offset on all redundancy nodes. For instance, the blocks on offset 1 (A1, B1, C1, P1, and Q1) form a redundancy ensemble, i.e. a set of coherent data and redundancy blocks. The main difference to the RAID scheme, however, is that the ClusterRAID does not stripe data over multiple devices. Logically consecutive data always resides on one disk. This is the main characteristic of the architecture and common to all ClusterRAID distribution schemes.
In order to overcome possible bottlenecks induced by the distinction between data and redundancy nodes, the redundancy information can also be distributed over all nodes. An example of such a RAID-5 like distribution is shown in figure 3.3. All nodes store user data as well as redundancy information. In order to determine the mapping between the data and the redundancy blocks, it is

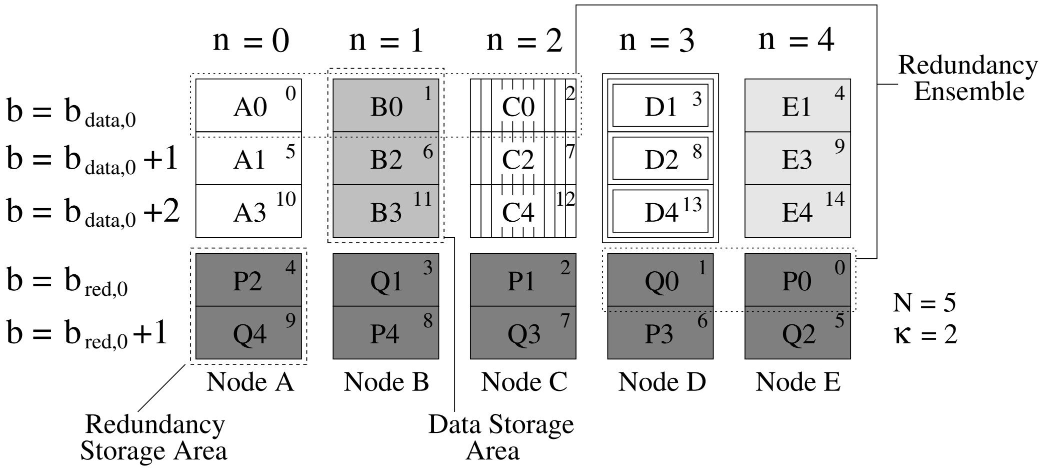
n = 0
n = 1
n = 2
n = 3
n = 4
Redundancy Ensemble
b = b data,0
b = b data,0 +1
b = b data,0 +2
b = b red,0
b = b red,0 +1
A0 0
B0 1
C0 2
D1 3
E1 4
A1 5
B2 6
C2 7
D2 8
E3 9
A3 10
B3 11
C4 12
D4 13
E4 14
P2 4
Q1 3
P1 2
Q0 1
P0 0
Q4 9
P4 8
Q3 7
P3 6
Q2 5
N = 5
κ = 2
Node A
Node B
Node C
Node D
Node E
Redundancy Storage Area
Data Storage Area

FIGURE 3.3.: Data distribution scheme with distributed redundancy information. All the disks (or nodes) A through E store both user data and redundancy information. Small numbers in the top right corner of each block denote their consecutive numbering (λ_{data},λ_{red}) used for the mapping relations. The block numbers $b_{data,0}$ and $b_{red,0}$ mark the beginning of the corresponding storage areas.

convenient to introduce a consecutive numbering for all data blocks, all redundancy blocks, and the redundancy ensembles. These numbering will be denoted by λ_{data}, λ_{red}, and λ_{ens}, respectively. The consecutive number λ_{data} of a data block can be calculated from the block's ID tuple $\{n,b\}_{data}$ by

$$\lambda_{data} = Nb + n \tag{3.4}$$

The number of the redundancy ensemble of this block is determined by dividing the consecutive block number λ_{data} by the number of data blocks per redundancy ensemble:

$$\lambda_{ens} = \left\lfloor \frac{\lambda_{data}}{N-\kappa} \right\rfloor \tag{3.5}$$

The number λ_{red} of the first redundancy block assigned to a data block can then be calculated from

$$\lambda_{red} = \lambda_{ens}\kappa \tag{3.6}$$

The consecutive number of the redundancy block can now be mapped backwards to a node ID and a block offset in the redundancy storage area

$$\{n,b\}_{red} = \{N - 1 - \lambda_{red} \bmod N, \left\lfloor \frac{\lambda_{red}}{N} \right\rfloor\} \tag{3.7}$$

The numbers of the other $\kappa - 1$ redundancy blocks assigned to this data block are given by the numbers $\lambda_{red} + 1, \lambda_{red} + 2, \ldots, \lambda_{red} + \kappa - 1$, which can easily be converted back into node IDs and block offsets using equation 3.7.

Similar relations can be used to determine the remaining data nodes and blocks in case of a data device failure. Given the consecutive number λ_{data} the expression

$$\lambda_{data} \bmod (N-\kappa) \tag{3.8}$$

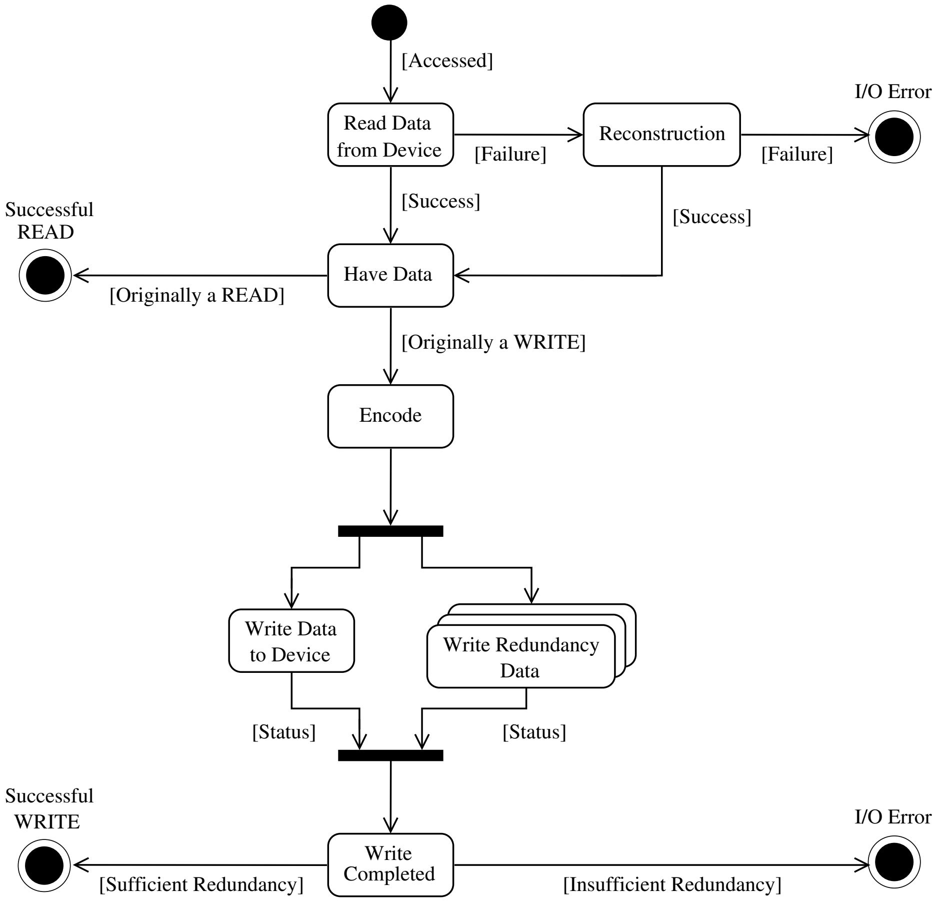
[Accessed]
Read Data from Device
[Failure]
Reconstruction
[Failure]
I/O Error
[Success]
[Success]
Successful READ
Have Data
[Originally a READ]
[Originally a WRITE]
Encode
Write Data to Device
Write Redundancy Data
[Status]
[Status]
Successful WRITE
Write Completed
[Sufficient Redundancy]
[Insufficient Redundancy]
I/O Error

FIGURE 3.4.: UML activity state diagram of the ClusterRAID when serving requests. All accesses to the ClusterRAID require to retrieve the corresponding block from the device.

gives the number of the data block within the redundancy ensemble and, thus, the other $N-\kappa-1$ consecutive numbers by addition and subtraction. The relation

$$\{n, b\}_{data} = \{\lambda_{data} \bmod N, \left\lfloor \frac{\lambda_{data}}{N} \right\rfloor\} \tag{3.9}$$

can then be used to determine the node ID and block offsets of these data blocks. The backward mapping from redundancy blocks to their associated data blocks can be conducted in direct analogy to the forward mapping shown in equations 3.4 through 3.7.

The distribution of redundancy blocks inevitably affects the performance and traffic pattern of a ClusterRAID system. Section 3.3 will return to this point and examine the two presented distributions in detail.

Since the ClusterRAID intercepts, controls, and supervises all disk transactions involving its underlying device, it also must check the status of the individual requests after being serviced. Figure 3.4 depicts a UML [113] diagram of the activities of the ClusterRAID while serving requests. Independent from whether the current request is a read or a write, the corresponding block is first retrieved from the underlying device. If the request is a read and the disk transaction succeeds, the

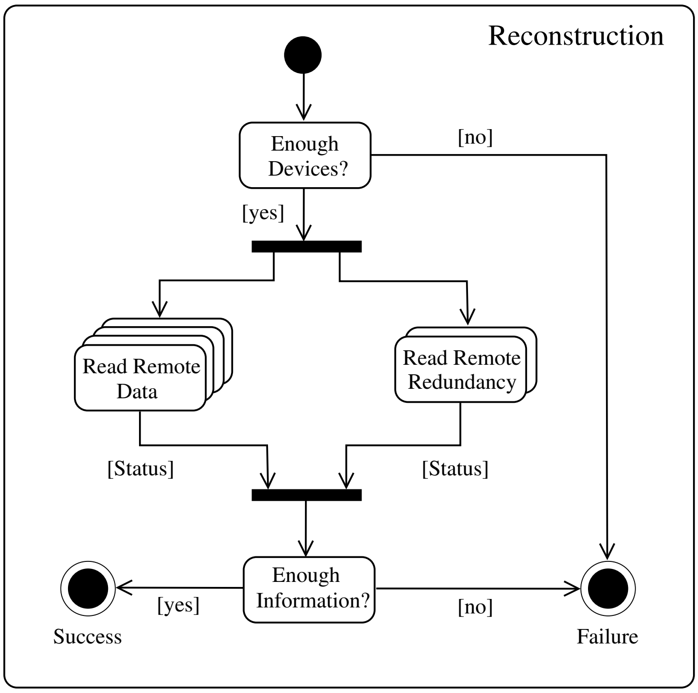

FIGURE 3.5.: UML activity subdiagram for the reconstruction of data. Additional device failures during reconstruction are reported to the ClusterRAID system by the completion status information of the issued requests. If the number of failed devices does not exceed the threshold as given by the algorithm, the data can be reconstructed.

ClusterRAID returns the data back to the requester and the transaction is completed. A successful read is the most common scenario and the ClusterRAID architecture is therefore optimized for this case.

If the original request was a write, the difference between the data just read and the data to be written is the basis for the calculation of the redundancy information. As will be shown in section 4, an update of the redundancy information can be calculated from the difference between the data already stored in a block and the data about to be written to it. The import of the contents of the other data blocks sharing that redundancy block is not required. This difference is encoded and sent to each remote redundancy storage area. If all these writes are successfully completed, the original write request is signaled as completed to the requester as well.

If an error occurs during the initial read of a data block from the local device, the data can be reconstructed using the redundancy information stored for this block on the assigned redundancy blocks together with the data blocks of the same redundancy ensemble. Figure 3.5 shows the corresponding activity subdiagram. After checking the number of operational devices in order to determine if sufficient data is available for the reconstruction process, read requests are issued to the remote data and the redundancy devices. If all data for a particular block is gathered, the data is reconstructed. Of course, additional device failures can occur after the requests have been issued to the remote devices. The ClusterRAID is notified about the actual state of the individual devices by the request completion status. As long as the total number of failed devices does not exceed the limit given by

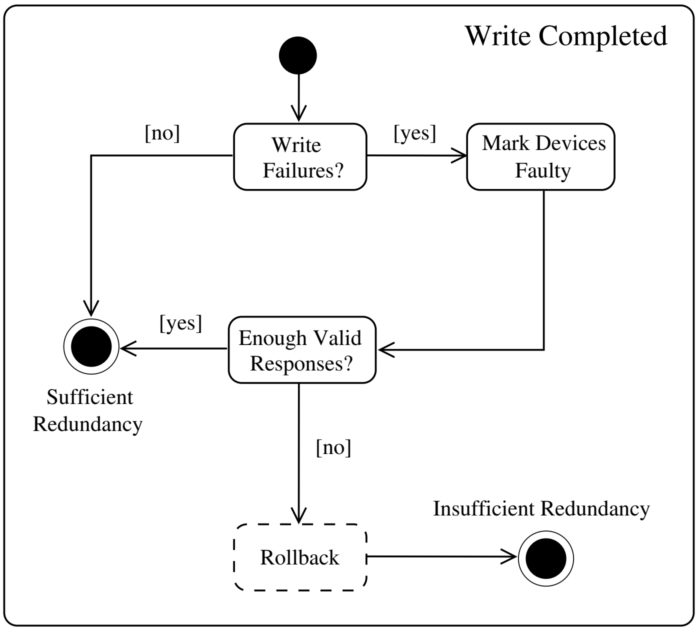

FIGURE 3.6.: UML activity subdiagram for the write completion. The rollback activity is marked by a dashed box indicating that it is not implemented for but would represent a reasonable extension of the prototype described in chapter 6.

the deployed algorithm, the data can be successfully reconstructed. Once this threshold is reached, the reconstruction must be aborted with a failure. If the reconstruction succeeds, the data is returned to the requester; if it fails, the request is ended with an I/O error.

A reconstruction can also take place if the original request was a write. After a successful reconstruction, the data flow is the same as for a successful preceeding read. Failures can, of course, also happen during the write of the data or the redundancy blocks. If the ClusterRAID uses an error-correcting code that is able to handle multiple device failures, the write request can be regarded as completed if the number of successful writes reaches a certain user-defined threshold. This threshold reflects the acceptable lower limit for the reliability of the data. Figure 3.6 depicts the activity diagram for the write completion. If no errors have occurred during the writing of the redundancy information the write transaction is regarded as completed. Otherwise, every device that has reported a failure during the write transaction is marked faulty in order to avoid submitting subsequent requests to it. If the status of all issued requests is known, the number of successful write transactions is compared to the above mentioned threshold. A sufficient number of successful transactions results in the whole transaction being regarded as successfully completed. If the threshold is not reached, the successful write transactions must be rolled back in order to preserve the consistency of data and redundancy information. The completion status of the overall transaction is then regarded as failed.

If a node or a device with a redundancy storage area encounters a repairable failure *during the update* of a redundancy block, the status of the block after the node or the device has been restored to health is not well-defined. The update may have been performed, i.e. the block incloses information about the new data, or the update was aborted and the content of the block is unchanged. This scenario introduces the danger of a spontaneous loss of consistency of data and redundancy information: node failures during redundancy block updates may leave a mixture of updated and unchanged redundancy blocks within a single redundancy ensemble. This inconsistency inhibits the reconstruction in case of additional device failures. To prevent this situation a conservative approach is to perform a full resynchronization of the repaired device or node by regenerating all redundancy information. Logging the numbers of lately updated blocks or regions could reduce the effort when doing so. A more elegant approach, however, would be to implement a two-phase commit protocol [89], which avoids inconsistencies from the outset. Keeping the update requests tagged with the ID of the originating node and a timestamp would allow to rollback updates on the other redundancy blocks in the ensemble in the event of such failures.

The above discussion only holds true for the assumption that the data node is notified about the completion status of the redundancy write, in particular of the failure. However, it may also happen that the redundancy node acknowledges the redundancy update before it is committed physically by the underlying device. A non-catastrophic failure in the time window between the acknowledgement and the physical update of a block would introduce a inconsistency between data and redundancy information. Disk manufacturers ensure that modern devices have enough capacitive energy to commit all write requests in the disk cache when spinning down due to a unscheduled shutdown, such as a power outage. This avoids the inconsistency. A similar problem may arise if the block update is acknowledged before it is sent to the device. However, since the ClusterRAID does not use any buffering on the redundancy side due to consistency reasons – this is discussed in detail in section 5.4.3 – this situation cannot occur. The acknowledge is always sent after the requests has been sent to the device.

The reconstruction of redundancy information only makes sense when the failed redundancy device is repaired and the reconstructed redundancy information is stored thereon. On demand reconstruction of redundancy information – regardless of the data node having issued a redundancy update or a read request in order to reconstruct its data – does not increase the amount of valid information in the system, but produces additional traffic instead.

Although the ClusterRAID can handle the loss of multiple devices or nodes, faulty components should be replaced as soon as possible, since the performance and the remaining failure resilience will be reduced if the ClusterRAID operates in degraded mode. The data of a faulty device can be reconstructed on any node. Hence, faulty nodes or nodes with faulty components can also be replaced by spare nodes which take over the replaced node's identity in the ClusterRAID network.

3.3. Performance

For local disks, read and write accesses are in general approximately equivalent in terms of complexity and performance: however, they are completely different in a ClusterRAID system. This is mainly due to the fact that write accesses entail a set of complex processing steps, as set out at the beginning of this chapter. On the other hand, the overhead for reading is kept as small as possible, as the ClusterRAID is optimized for the latter, more frequent case. Successful read accesses can be processed locally and require no interaction with or information from other nodes. Thus, the read performance available to a ClusterRAID node is very close to – and also limited by – the performance of the underlying disk. The aggregate read throughput is the sum of the disk bandwidths in the nodes with data storage areas. Obviously, this is also the theoretical upper limit. Since the ClusterRAID introduces only a small additional layer between an application and its storage device, the values for latency and computing overhead are also very close to that of a standard disk access. Furthermore, for successful reads the network is not involved at all.

In contrast with reading, the available write performance of a ClusterRAID node can be very different from the write performance of the constituent disk. The calculation of the redundancy information is required for every write request and increases the processor load. The redundancy information must be transferred to remote nodes and thus increases the load on the network as well as on the involved CPUs. The write latency is also increased compared to standard disk accesses, since the sharing of redundancy blocks with other nodes will require a certain amount of synchronization and locking. For the throughput, however, the disk transfer rate is the main limiting factor, since the transfer rates of gigabit networks surpass the throughput of single disks. The maximum throughput for writing is always limited to about half of the local disk speed, since every write requires a preceeding read to determine the difference between the old and the new block content. This is valid for both data and redundancy requests. In addition, a redundancy block is shared by multiple nodes. If several nodes access corresponding data blocks at the same time, the available bandwidth is reduced according to the number of simultaneous writers. In the following, performance considerations for write accesses will include both configurations, with dedicated and with distributed redundancy. It should be noted that these considerations are based upon simplified and conservative assumptions. Hence, the results derived from these considerations using realistic numbers for network bandwidth or disk transfer rates are mainly intended as an illustration and do not necessarily represent actual performance boundaries. The end of this section will give a first outlook on how the write performance can be improved by approaches beyond this first order approach.

In a configuration with dedicated redundancy nodes, the write bandwidth b_{write} available to a node depends on the local disk bandwidths of the data and the redundancy nodes b_{disk}^{data} and b_{disk}^{red}, the network bandwidth b_{net}, the number of concurrently writing nodes c, and the number of redundancy blocks κ to be written for any given data block. For reasons of simplicity it is assumed here that all nodes use the same type of interconnect and that the network infrastructure itself, e.g. the switch backplanes, present no limiting factor. The available bandwidth is then determined by the minimum of four terms

$$b_{write} = min\left\{\frac{b_{net}}{\kappa}, \frac{b_{net}}{c}, \frac{b_{disk}^{data}}{2}, \frac{b_{disk}^{red}}{2\,c}\right\} \tag{3.10}$$

The first contribution in this equation, b_{net}/κ, reflects the fact that a writing node has to transfer the redundancy information to all redundancy nodes. Thus, the fraction of the network bandwidth available to each redundancy node limits the maximum transfer rate. Similarly, the concurrent writers share the input bandwidth to the redundancy nodes, hence the term $\frac{b_{net}}{c}$. The remaining two terms, $b_{disk}^{data}/2$ and $b_{disk}^{red}/(2c)$, indicate the reduced bandwidth due to the required read-modify-write cycles upon writing. Ignored effects, such as the rotational delay when accessing the same block two times during a redundancy update, may further reduce the performance. Typically, the limiting factor is the disk, the network is only limiting the overall throughput if a low performance network such as Fast Ethernet is used.

Figure 3.7 shows the limits of the available write bandwidth per node as a function of the number

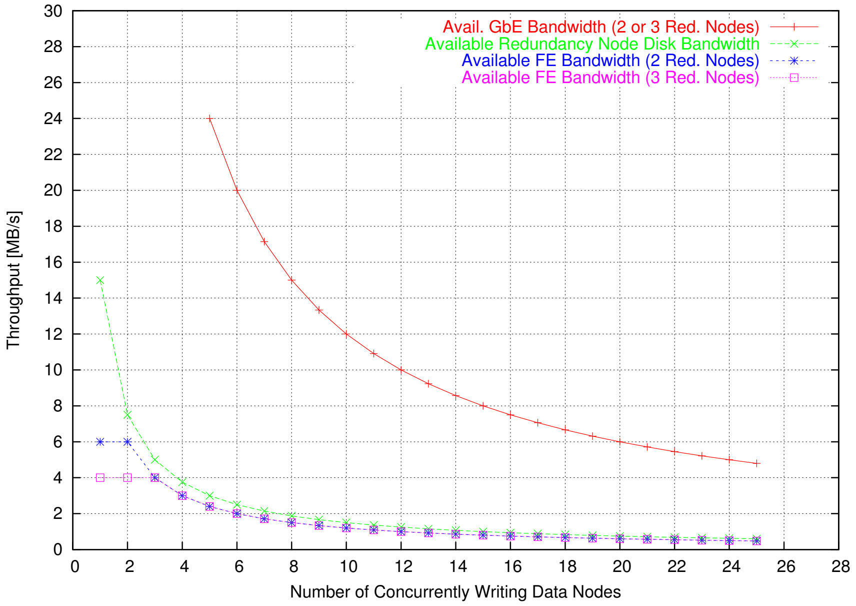

FIGURE 3.7.: Available write bandwidth for a ClusterRAID configuration with dedicated redundancy nodes. In this example, the bandwidth to the local disk is 30 MB/s on all nodes, and the network bandwidths are 12 MB/s and 120 MB/s for Fast Ethernet and Gigabit Ethernet, respectively.

of concurrent writers for the following assumptions: a disk bandwidth of 30 MB/s for both data and redundancy nodes, a network bandwidth of 12 MB/s for Fast Ethernet and 120 MB/s for Gigabit Ethernet, and $\kappa = 2$ or $\kappa = 3$. If Fast Ethernet is used, the network is always the limiting factor. As long as the number of writers is not larger than the number of redundancy nodes, the maximum available bandwidth is a constant, due to the first term in equation 3.10. If c is larger than κ, the bandwidth decreases with b_{net}/c. As mentioned above, the disk bandwidth is the limiting factor if Gigabit Ethernet is used. The available bandwidth decreases with $b_{disk}/(2c)$ in this scenario. Therefore, for the configuration with dedicated redundancy, the bottleneck of the write performance is given by the shared resources to and on the nodes that store the redundancy information. This is very similar to the bottleneck situation for the classic RAID-4 as discussed in section 2.1, where one parity device must be accessed for all data updates.

If the redundancy information is distributed over all nodes in a ClusterRAID configuration, the correlation between the available write bandwidth and the number of concurrent writers is not as obvious as for the dedicated redundancy setup. Since all nodes now store redundancy information in addition to their data, it is intuitively clear that the available bandwidth for writing should be increased when compared to a setup with dedicated redundancy nodes. As outlined in the previous sections, every block is assigned κ redundancy blocks, which must all reside on different nodes. If there are multiple writers, a node may have to store redundancy information from other nodes in addition to its own data. The exact number of blocks, however, depends on the specific combination of nodes and data blocks. In the best case all blocks to be written, data and redundancy, are equally distributed over all nodes in the ClusterRAID system. In the sample distribution of figure 3.3, simultaneous write accesses to the blocks $A0$, $B2$, $C4$, $D1$, and $E3$ represent such a node and block combination with equally distributed write loads, namely three blocks per node (one data block and two redundancy blocks). If, however, the blocks $A0$, $B2$, $C4$, $D2$, and $E4$ are written, node A ($n = 0$) receives five block updates, while node C ($n = 3$) receives only one block update. The maximum number of blocks that one of the nodes needs to update for such a node/block combination is taken as a first order approximation of the available write bandwidth. For a configuration with N nodes, κ redundancy blocks per data block, and c concurrent writers there are

$$\binom{N}{c}(N-\kappa)^c \tag{3.11}$$

node/block combinations to consider. The binomial coefficient is the selection of the c writing nodes out of the N total nodes. The second term is the number of block combinations that could be written on the c selected nodes. Only the first $N - \kappa$ blocks on each node are taken into account, since their combinations form the equivalence classes of all possible block combinations with respect to the redundancy block mapping function. The total number of blocks that are written in one time step in the whole ClusterRAID system is

$$c(1+\kappa) \tag{3.12}$$

Hence, the theoretical lower limit of maximal block writes per node for an equal distribution is

$$\left\lceil \frac{c(1+\kappa)}{N} \right\rceil \tag{3.13}$$

and the theoretical upper limit is c, a local data write and redundancy information from all other nodes. The worst case scenario of c write accesses on one node corresponds to the standard case for a ClusterRAID with dedicated redundancy. Figure 3.8 shows the distribution of the maximum number of blocks that have to be written for a setup with $N = 10$ nodes, $\kappa = 2$, and $c = 10$ concurrent writers when all block and node combinations are taken into account. At the lower end the distribution starts at three blocks, since a total of 30 blocks must be written to 10 nodes. If the blocks are equally distributed, the maximum number of blocks to be written per node is thus 3. In this specific example there are no node and block combinations in which all writers map one of their blocks to a single node. Hence, there is no entry at 10 in the histogram. The centroid of this distribution gives an approximation of the average slowdown for writing in a ClusterRAID with distributed redundancy information and this specific configuration.

Along with the theoretical boundaries, the centroids of the histograms for a varying number of writers for this specific setup are depicted in figure 3.9. For the worst case the maximum number of hits scales directly with the number of concurrent writers. The steps in the best case graph are due to the ceiling rounding in equation 3.13. On average, the distribution of redundancy information will result in an improved write performance compared to a configuration with dedicated redundancy. The expected throughput deduced from this plot is shown in figure 3.10. Of course, the throughput also decreases with the number of concurrent writers, but is on average much better than $1/2c$.

As already mentioned, these considerations are simplified. Redundancy information and data reside on a single device. Hence, local data updates and redundancy updates from remote nodes affect the bandwidth available to the other access type. This effect could, for instance, be reduced by using independent devices for data and redundancy. For redundancy writes, coalescing, i.e. the merging of

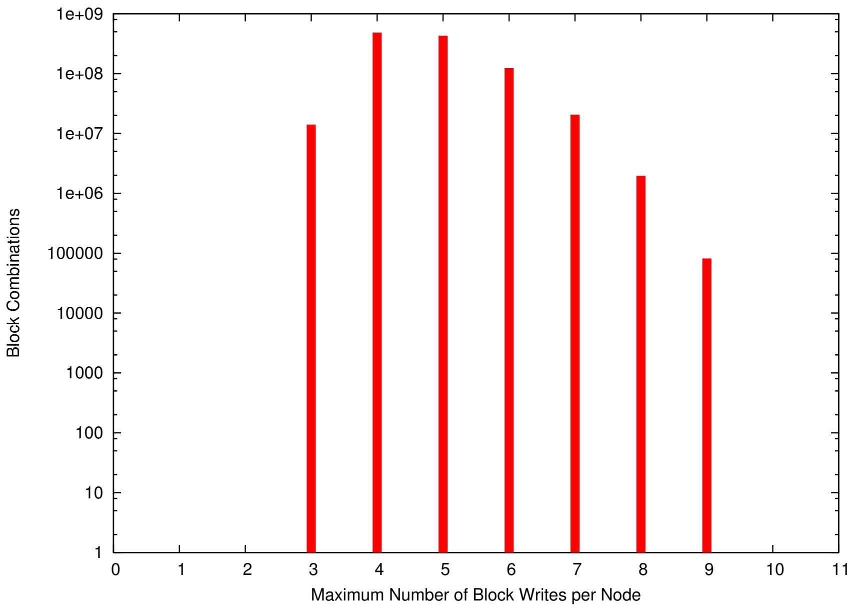

FIGURE 3.8.: Distribution of the maximum number of block writes on a single node for a ClusterRAID with $N = 10$, $c = 10$, and $\kappa = 2$.

requests to the same block into a single disk access, can also reduce the average number of blocks to be written per device. This concept is very similar to that of the buffer cache in the Linux kernel: delaying and waiting for other requests to the same block may improve the write performance [114]. In the above simulation, every block to be written was considered to be independent from all other accesses to the device and thus reduced the available bandwidth. Further improvement of the write speed by orchestrated accesses of the ClusterRAID nodes to good block combinations, i.e. combinations where the redundancy information is equally distributed, may turn out to be impractical. Despite this, knowledge of the access pattern of a specific application may be used to define a mapping function that distributes accesses equally over the constituent ClusterRAID nodes. In addition, other approaches, such as lazy updates along with write barriers, as explained in section 2.4.2, could be used to increase the ClusterRAID's write throughput.

Since the ClusterRAID is able to cope with multiple simultaneous failures, a particularly fast reconstruction of a faulty device is not essential for its operation. However, the performance of the ClusterRAID is of course affected when faulty devices are present and nodes operate in degraded mode. Therefore, a rough estimation of the upper limit for the performance during the recovery of data shall close this section. For the reconstruction of a data block on a node with a faulty device at least $N - \kappa$ of the corresponding redundancy ensemble are required. The data reconstruction rate is then limited by the minimum of the network transfer rate and the aggregate disk read bandwidth:

$$b_{rec} = min\left\{ b_{disk}, \frac{b_{net}}{(N-\kappa)} \right\} \tag{3.14}$$

Blocks requested by applications can of course be reconstructed on demand – it is not necessary to wait for the whole device to be recovered. Hence, latency for block accesses in degraded mode

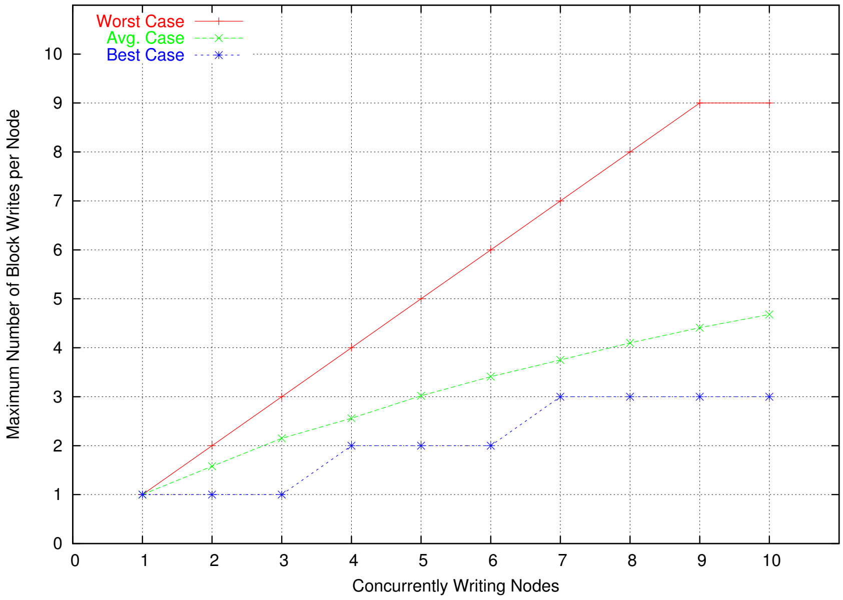

FIGURE 3.9.: Worst case, average case and best case of the maximum number of block writes for a distributed redundancy ClusterRAID setup with 10 nodes depending on the number of concurrent writers. The worst case scenario corresponds to the standard case for a ClusterRAID with dedicated redundancy.

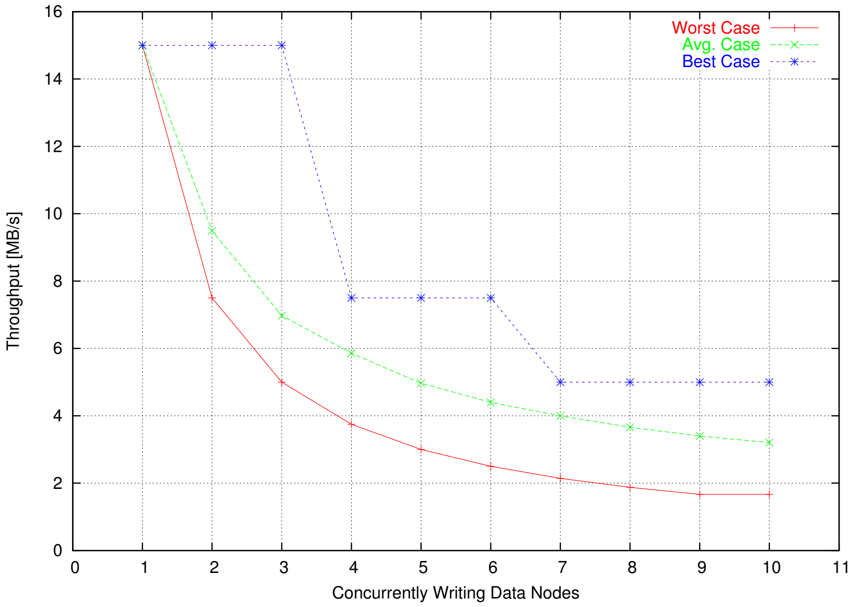

FIGURE 3.10.: Expected ClusterRAID write throughput for distributed redundancy. In this example the setup comprises 10 nodes, the disk bandwidth is 30 MB/s, and the Gigabit Ethernet network can transfer 120 MB/s.

Request Type	No. of Nodes w/ disk I/O	No. of I/O op. (total)	No. of Remote I/O op.
Read	1	1	0
Write	1+κ	2+2κ	2 κ
Degraded Read	G-κ	G-κ	G-κ
Degraded Write	G-1	G+κ	G+κ

TABLE 3.1.: Number of nodes involved and number of (remote) I/O operations required for a single block with each of the four different request types. The parameter G is the total number of nodes in a redundancy ensemble.

will depend on disk latency, network latency, and the time required to actually decode the data. It should be noted that formula 3.14 excludes concurrent read or write accesses on the remote nodes, and also neglects the computational overhead of the error-correcting code. In addition, if the data is reconstructed to a spare device the write speed of this device may also turn out to be a limiting factor.

3.4. Scalability

A system comprising completely independent nodes can scale to arbitrary size. The more the coupling of the nodes is increased to improve the system's overall performance, the more effort is required for synchronization and coordination to avoid mutual interference. In the ClusterRAID storage architecture the coupling of nodes is kept to a minimum. Read requests are served from the local device, asynchronously from any I/O operations on other nodes. Hence, in case of read operations the ClusterRAID resembles a collection of independent nodes. The coupling of the nodes is increased for writing. Redundancy information must be transferred to associated remote redundancy areas. This reduced independency inevitably impacts the scalability of the system. If a request is issued to a faulty device, the system operates in degraded mode. In order to serve read requests in degraded mode, the currently unavailable local data must be reconstructed from redundancy information and remote data by applying the inverse coding algorithm. Thus, an even closer coupling of the nodes is required. In addition to the reconstruction, write requests to a system in degraded mode also require the transfer of the redundancy information. The coupling between the nodes is closest in this last scenario.

Table 3.1 lists the four mentioned request types along with the number of nodes involved and the number of I/O operations needed to serve the corresponding request. The parameter G is the group size, i.e. the number of nodes in a redundancy ensemble, and κ is the number of redundancy areas per data block used in this group. Reading requires no interaction with other nodes and only a single I/O operation, i.e. the read request to the local device. In addition to the local node, writes involve all the nodes that host a redundancy area for the requested block. Every write is preceeded by a read, and thus the number of I/O operations is twice the number of involved nodes. In degraded mode, the ClusterRAID performs remote I/O operations only. In order to reconstruct the local data, the redundancy information and the data of other nodes in the group is required. For space optimal codes, at least G-κ nodes are involved. In addition to the reconstruction, new redundancy information must be determined and transferred to the corresponding κ redundancy areas for a degraded write. As this

requires additional 2κ I/O operations, the total number of I/O operations is G+κ.
The amount of concurrent accesses to the shared resources, i.e. network or redundancy areas, limits the scalability of the ClusterRAID. However, as stated in section 3.1, concurrent accesses, i.e. write requests, are rare compared to independent, i.e. read, requests in many applications. In a configuration with dedicated redundancy nodes, their disk bandwidth is shared by the concurrent writers and will present a bottleneck. The distribution of the redundancy information avoids this bottleneck and shows a better scaling. The amount of shared resources is a tradeoff between performance, space overhead, and reliability. If the total set of nodes is split into smaller redundancy groups with the same number of redundancy blocks per redundancy ensemble, the space overhead is increased, but the concurrency is reduced. This approach, for instance used by Swarm [115] or the Panasas ActiveScale File System [116], always allows to optimize the ClusterRAID with respect to the size of an installation and the application requirements.

3.5. Fault-tolerance

Fault-tolerance is one of the crucial aspects when building distributed mass storage systems. This is in particular true if commodity off-the-shelf components are used, since the quality of standard components is typically worse than that of high-end products. Several reliability issues influence the architecture of such systems and determine the degree of reliance including, amongst others: the lifetime and fault-tolerance of individual components, the time needed for their repair in case of a failure, the size of the system, and the different kinds of failures to occur. This section will give an overview of two simple models for estimating the lifetime of such systems. Both models are then applied to estimate the mean time to failure of a ClusterRAID system.

3.5.1. Basic definitions

The term *fault-tolerance* refers to the ability of a system to deliver a certain level of service in the event of a component failure. The four classes of fault-tolerance include:

- redundancy,
- fault-isolation,
- fault-detection and annunciation, and
- online repair.

Most systems provide fault-tolerance by redundancy: additional components are added in order to take over a service if the primary components fail. Examples are redundant power supplies, replicated data, additional network paths, or fail-over file servers. Fault-isolation refers to the ability of the system to limit the scope of the consequences resulting from faulty components. An example is the management of processes in an operating system: different processes are protected from each other, for instance by limiting the memory access, with the intention of reducing the impact that a process can have on other processes in the system. Fault-detection and annunciation include techniques such as predictive failure analysis, as for instance defined in the Self-Monitoring Analysis and Repair Technology (SMART) specification [117, 118], to detect and predict component failures and announce them to take appropriate actions before the failure occurs. Usually, this is achieved by monitoring device characteristics and alerting the host system if pre-determined thresholds are surpassed. Timely replacement of components that will fail in the near future increases the overall reliability of a system. Online-repair is the ability of a system to hide the effects of faulty components. Aside from a possibly degraded performance the system is fully functional. A RAID-5 system is a typical example of a system deploying online repair: if a disk fails, the requested data can be reconstructed on demand from parity information.

Often, the *Mean Time Between Failures (MTBF)* or the *Mean Time To Failure (MTTF)* is used as a parameter to describe a system's reliability. The term MTBF is used with repairable components, while MTTF is used with non-repairable components. The MTBF is the amount of time a system is operational without any failures. It is calculated on the basis of statistical models and thus has a more theoretical character. Typically tested on a large number of systems, the MTBF is the average number of hours a systems works before a failure occurs. It can be used to determine the reliability of a system. The *reliability R* is defined as the probability that the system will survive a certain amount of time without failure. The probability $R(t)$ that the system is still operational after the time t has elapsed and the system was operational at $t = 0$ is given by the exponential expression

$$R(t) = exp(-t\ r_f) \tag{3.15}$$

r_f is the inverse of the MTBF, called the failure rate. For instance, a disk with an MTBF of 500,000 hours has a 91.6% probability of working for 5 years without a failure.

Equation 3.15 assumes the failure rate r_f to be a constant. This is only true for a certain domain in a system's life span within which failures are randomly distributed over time. Electromechanical systems, such as hard disk drives, undergo different phases in which this assumption does not hold. Typically, these systems have a high initial failure rate in the burn-in phase, followed by a period of randomly distributed failures, before the system begins to wear out and the failure probability rises again. This varying reliability is usually described by Weibull's formula [119]:

$$R(t) = exp\left(-\left(\frac{t}{\eta}\right)^{\beta}\right) \tag{3.16}$$

The parameter β determines the shape of the reliability curve, while the scale parameter η is a measure for the characteristic lifetime in the particular phases. The *hazard rate h(t)* of a system is then given as the ratio of the failure probability density $f(t)$ to the reliability at that particular time

$$h(t) = \frac{f(t)}{R(t)} = \frac{1}{R(t)}\frac{d}{dt}(1 - R(t)) = \frac{\beta}{\eta}\left(\frac{t}{\eta}\right)^{\beta-1} \tag{3.17}$$

The shape parameter β reflects the failure mechanisms predominant in the current phase. According to the discussion above, the hazard rate for electromechanical systems often follows the "bathtub form", as shown in figure 3.11. The actual value of β, and hence the hazard rate, is of course dependent on a number of other parameters, such as the environmental temperature, the number of on/off cycles, the read/write profile, the vintage, the device manufacturer, or the system configuration. Therefore, and since disk manufacturers publish no detailed data on their reliability analysis, it is not possible to apply a detailed Weibull analysis in this section. Although the MTBF only allows

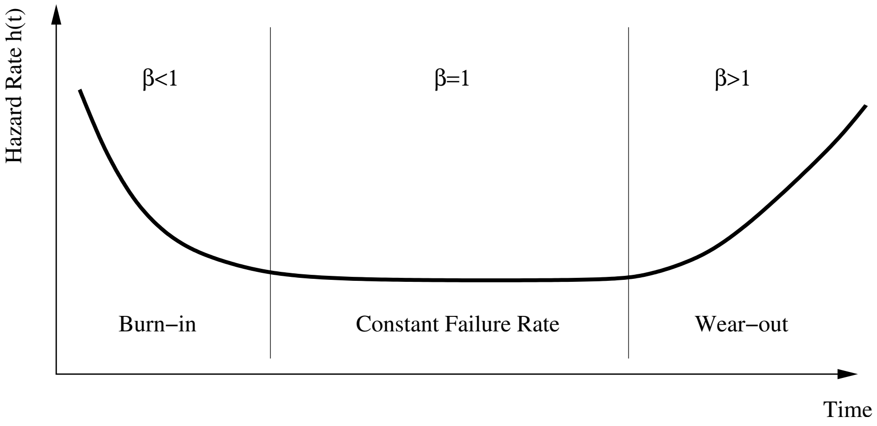

FIGURE 3.11.: Schematic of a Weibull distribution. The shape parameter β reflects the dominant failure mechanisms.

for a somewhat coarse estimation, it is a widely accepted measure for the reliability of a system and will therefore be used in the following discussion.

The *availability* A of a system is the percentage of time in which the system's service is available for use. It is usually defined by

$$A = \frac{uptime}{uptime + downtime} \tag{3.18}$$

where $uptime$ and $downtime$ stand for the time intervals in which a system is available or not available, respectively. While equation 3.18 considers all downtimes, including, for example, system abuse or maintenance, the *inherent availability* A_i, defined by

$$A_i = \frac{MTBF}{MTBF + MTTR} \tag{3.19}$$

covers only the downtimes for system repair, represented by the *Mean Time To Repair (MTTR)*. Thus, A_i is the maximal achievable availability.

When the reliability of mass storage systems is discussed, the MTBF or the MTTF is often replaced by the *MTTDL*, the *Mean Time To Data Loss*. The MTTDL reflects the point in time at which the system has irrecoverably lost part or all of the data due to one or more component failures. Another important concept in this context is the distinction between *errors* and *erasures*: an erasure is an error whose location is known. In case of disk failures the location of the error, the faulty disk, is usually known. Chapter 4 will return to this topic and use the location information to reduce the complexity of error correction codes.

3.5.2. Reliability Modeling

In the original RAID paper a formula was derived to estimate the MTTDL for disk arrays [66]:

$$MTTDL_{RAID} = \frac{MTTF_{disk}}{N} \times \frac{MTTF_{disk}}{(G-1)MTTR_{disk}} \tag{3.20}$$

In this equation, $MTTF_{disk}$ is the mean time to failure of a single disk, N is the total number of disks in the system, G is the number of disks in a stripe group, i.e. the number of disks to be accessed during reconstruction, and $MTTR_{disk}$ is the mean time to repair of a faulty disk. For a system with N components the ensemble's MTTF is given by the MTTF of the components divided by N, hence the first term in 3.20. The second term reflects the fact that another failure of one of the remaining $G-1$ devices in the stripe group during the repair of the first failure will result in data loss.

The cited paper only considers parity as the redundancy algorithm and hence only a protection against single failures. In a subsequent publication, equation 3.20 was extended for algorithms that are able to tolerate κ failures [110]:

$$MTTDL_{RAID} = \frac{MTTF_{disk}}{N\prod_{i=1}^{\kappa}(G-i)}\left(\frac{MTTF_{disk}}{MTTR_{disk}}\right)^{\kappa} \tag{3.21}$$

The appearance of equation 3.21 has been chosen on purpose: it emphasizes the benefit of deploying algorithms that are able to correct more than a single error. The MTTDL$_{RAID}$ is multiplied by the ratio of MTTF$_{disk}$ and MTTR$_{disk}$ for every additional error the system can cope with. The MTTR of a modern disk is usually of the order of several hours, while its MTTF is between a half and one million hours [17]. The repeated division by the group size is therefore small compared to that factor. A simple example will serve as an illustration. Considering an ensemble of 1,000 disks, organized as RAID-5 arrays with five disks each, and assuming each disk has an MTTF of 500,000 hours and the MTTR of a disk to be 24 hours, the MTTDL of the whole system is almost 297 years.[2] Hence, the probability that all the data is still available after a ten years period is 96.7%. If the disks are organized in an array with 20 ensembles of 50 disks each, where three of these 50 disks are used to store redundancy information and the algorithm used is able to cope with three simultaneous failures in an ensemble,[3] then the MTBF of the whole system is more than 4,600,000 years. The probability to have all data still available after ten years is beyond 99.999%. Since, in addition, the space overhead for redundancy data is 20% for the RAID-5 arrays and only 6% for the second configuration, this example shows the advantages of sophisticated redundancy algorithms in terms of the MTBF and space compared to simple, parity-based schemes.

This model is simplistic, however. It only considers failures of completely independent disks and neglects other effects, such as correlated disk failures, uncorrectable bit errors, or system crashes. Correlated disk failures can have several causes: the disks in a system may be exposed to similar stress, power supply failures, several disks belong to a batch with a systematic manufacturing defect, or the failures occur during the burn-in phase or when the disks begin to wear out. One way to model the correlation of disk failures is to reduce the MTTF of the disks in the system by a certain factor for any additional failure. The model given in [110], for instance, reduces the MTTF by a factor of 10 leading to the following extension of equation 3.21:

---

[2] This and all following illustrations will use sample numbers for the lifetime of disks as given by disk manufacturers. Although these numbers are derived from testing (under optimal environmental conditions though) and the application of complex statistical methods, it should be noted none the less that manufacturers are of course interested in high reliability figures for their products. As also discussed in section 1.1.1, field experience shows that the failure probability can be much higher than the one indicated by a disk's MTTF. Therefore, the relative proportions rather than absolute numbers are important in the reliability discussions throughout this chapter.

[3] Section 4 will show that such algorithms exist.

$$MTTDL_{RAID} = \frac{MTTF_{disk}}{N\prod_{i=1}^{\kappa}(G-i)}\left(\frac{MTTF_{disk}}{10 \times MTTR_{disk}}\right)^{\kappa} \tag{3.22}$$

Disk manufacturers claim that modern disks have bit error rates (BER) between 10^{-14} and 10^{-16} [120, 121, 122, 123, 124, 125]. Since 512 Bytes is a typical sector size, the probability to successfully read s sectors is thus

$$p_{BER}(s) = \left(1 - 512 \times 8 \times 10^{-15}\right)^{s} = \left(1 - 4096 \times 10^{-15}\right)^{s} \tag{3.23}$$

for a BER of 10^{-15}. Hence, approximately every 100 terabytes of reading a sector is irretrievable.

The third factor to be considered are system crashes. If the consistency of data and redundancy information is not protected by additional measures, such as atomic commits and rollbacks as used in databases, node crashes during a write operation may leave the system in an inconsistent state. Any additional failure of a data device results in a loss of data. A suitable term to describe the MTTDL by system crashes for this simplistic assumptions would be

$$MTTDL_{sys} = \frac{MTTF_{sys}}{N} \times \frac{MTTF_{disk}}{MTTR_{sys}} \tag{3.24}$$

This term is independent of the number of tolerable errors, since even in the case where the data is protected by multiple checksums, their state may be inconsistent and thus unusable after a crash.

It has been shown in [110] that the calculated MTBF of a RAID-5 array can be reduced by more than two orders of magnitude if all these factors are taken into account.

3.5.3. The Reliability of the ClusterRAID

In reference to the reliability classification presented in section 3.5.1, the ClusterRAID architecture provides fault-tolerance by fault-isolation, redundancy, and online repair.

Since data is always private to the local host, a device failure does not affect the data integrity of other nodes. Any device failure is isolated to the node on which it occurs. Even in the unlikely case that the number of failures should exceed the tolerance threshold of the redundancy algorithm being used, the data on the remaining nodes is unaffected. This is a major difference to architectures based on striping and parity, where the loss of more than one device always results in a total loss of data. In addition, the ClusterRAID stores all data in a redundant fashion, using sophisticated error-correcting codes to protect from data loss. The generated redundancy information can be used to reconstruct data in case of device failures. Reconstruction of data can also be carried out online: the decoding of the redundancy information is triggered by a request for a block on a failed device, and that request is served immediately. Hence, the local ClusterRAID device can be used as before, without interrupting uptime. A degradation in performance, however, is expected due to the decoding of the redundancy information.

Figure 3.12 shows the ClusterRAID MTTDL by disk failure for an example cluster with 1,000 nodes, calculated using equation 3.21. The cluster is divided into groups, sets of nodes that share redundancy information. The three graphs represent different thresholds of tolerable disk failures before data loss. The MTTDL *of the whole cluster* for a group size of 25 nodes and a redundancy algorithm that can cope with 3 disk failures is about 42,000,000 years. The probability of loosing no data

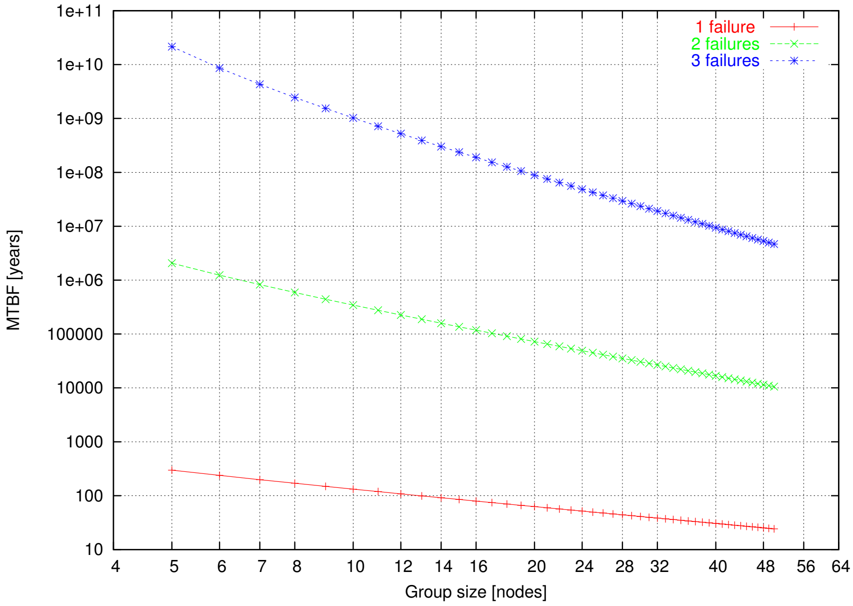

FIGURE 3.12.: ClusterRAID MTBF for disk failures. The total number of nodes is 1,000. The MTBF of the disks is 500,000 hours, the MTTR is 24 hours. The group size is the number of nodes in the independent redundancy groups.

within a 10 years period is thus more than 99.999%. The probability of loosing data by disk failure within a *single group of 25 nodes* protected by 3 redundancy nodes is below 10^{-8}.
However, as indicated in the previous section, further factors must be considered to obtain a more realistic picture of a system's reliability. As for local RAID systems, the hard drives' bit error rate and node failures, e.g. resulting from operating system crashes or power outages, can also reduce the MTTDL of a ClusterRAID system if no additional measures are taken. Since the system is distributed, failures of network interconnects or switches may also influence its reliability. Moreover, combinations of these failures must also be considered. In order to estimate the ClusterRAID's reliability, however, it is sufficient to consider the most probable combinations of failures:

- multiple disk failures,
- an operating system crash and a disk failure,
- an operating system crash and a faulty block due to the bit error rate, and
- multiple disk failures and a faulty block due to the bit error rate

Multiple operating system crashes can be neglected, since the first crash is assumed to already leave the redundancy information and the data in an inconsistent state. Additional node failures cannot worsen the situation. Since bit errors are isolated to a single block and the probability that the exact same block is affected on multiple devices is very small, data loss by multiple faulty bits is also not considered. Since network failures that affect the data integrity are very rare, they are also neglected in the following calculation.

For multiple disk failures equation 3.21 can be used. If one of the nodes has crashed leaving the redundancy information inconsistent, a single additional failure will lead to data loss:

$$MTTDL_{sys,disk} = \frac{MTBF_{sys}}{N} \times \frac{MTTF_{disk}}{(G-1) \times MTTR_{red}} \times \frac{1}{p_w} \tag{3.25}$$

In this term, the mean time between system crashes of a node is represented by $MTBF_{sys}$ and the mean time to resynchronize the redundancy information by $MTTR_{red}$. $MTTF_{disk}$, N, and G have the same meanings as above. The parameter p_w reflects the probability that a write was pending or in progress during the system crash, and hence the probability that the system crash leads to an inconsistent state.

Similarly, a bit failure after a system crash leads to the following term

$$MTTDL_{sys,BER} = \frac{MTBF_{sys}}{N} \times \frac{1}{(1-p_{BER}(s))^{G-1}} \times \frac{1}{p_w} \tag{3.26}$$

The bit error rate is taken into account by p_{BER}, the probability to encounter no bit error in a specific number of sectors, see equation 3.23. The number of sectors here is the amount of data to be retrieved from any of the $G-1$ devices in the group.[4]

Finally, the combination of multiple disk failures and a bit error leads to

$$MTTDL_{disk,BER} = \frac{MTTF_{disk}}{N\prod_{i=1}^{\kappa-1}(G-i)} \left(\frac{MTTF_{disk}}{MTTR_{disk}}\right)^{\kappa-1} \times \frac{1}{(1-p_{BER}(s))^{G-(\kappa-1)}} \tag{3.27}$$

The harmonic sum of the contributions from equations 3.21 and 3.25 through 3.27 provide an approximation of the overall MTTDL of the ClusterRAID.

Some sample numbers illustrate the introduced contributions. For the system characteristics summarized in table 3.2, the contributions of the above terms to the MTTDL of a ClusterRAID system are given in table 3.3.

The adoption of error codes that can to tolerate multiple device failures reduces the probability of data loss by disk failures to an acceptable level. This is necessary, since recent surveys of the reliability of nodes in large commercial-off-the-shelf clusters indicate that the disks are indeed by far the most frequent cause of failures [127].[5] Since the bit error rate of available hard drives has not significantly improved over the past few years, whereas capacity is doubled every year, unrecoverable bit errors more and more become a significant source of errors. Techniques such as predictive failure analysis should be deployed in order to detect wear-out effects and to identify devices that are expected to fail soon as early as possible. This can reduce the failures through bit errors significantly [130]. In this context, it should also be noted again that the variance of the failure probability over time is not reflected in the MTBF as the basis of this discussion. Therefore, a reliability centered maintenance strategy which takes the hazard rate over time into account, i.e. the burn-in and the wear-out phase, could help improve the overall system's reliability significantly.

After disk failures, operating system crashes are the largest reliability problem for the ClusterRAID,

---

[4] It should be noted here once more, however, that the contributions of the equations 3.25 and 3.26 only hold if no additional measures to prevent inconsistencies by system crashes, such as the implementation of a two-pase commit protocol, are taken into account.

[5] Older investigations report the hard disks to be the most reliable component in a cluster [128]. One of the reasons for these contrary results may be the type of disks used in the nodes (IDE vs. SCSI), another may be a decline of hard drive quality over the last years [129].

$MTTF_{disk}$	$500,000$ hours
$MTTR_{disk}$	24 hours
$MTTF_{sys}$	$250,000$ hours
$MTTR_{red}$	24 hours
Disk size	1 TB
Bit error rate	10^{-15}
p_w	0.1
κ	2
Group size G	25
Total number of nodes N	$1,000$

Table 3.2.: Reliability parameters used for the example calculation. While the $MTTF_{disk}$ of 500,000 hours is a typical value as stated by disk manufacturers, the $MTTF_{sys}$ is based on experiences with large clusters [126].

Failure combination	MTTDL	Lost data (abs.)	Lost data (rel.)	Probability of data loss in a 10 years period
Triple disk failure	42.2×10^7 years	3 TB (three disks)	0.003	$< 10^{-8}$
Double disk failure plus bit error	5.4×10^4 years	$1,536$ B (three sectors)	$\approx 1 \times 10^{-12}$	0.0002
System crash plus disk failure	5.9×10^3 years	1 TB (one disk)	0.001	0.001
System crash plus bit error	0.33 years	512 B (one sector)	$\approx 5 \times 10^{-13}$	≈ 1

Table 3.3.: Contributions of the individual failure combinations to the reliability of a ClusterRAID system for a cluster with 1,000 nodes and reliability parameters as given in table 3.2. The two bottom rows including system crashes are only valid if no additional measures for data integrity are incorporated.

since they may undermine the protection against multiple failures. Although not yet implemented in the ClusterRAID prototype discussed in chapter 5, the impact of system crashes must be eliminated in the long run. This can be achieved by adopting well-known techniques such as two-phase commits and rollbacks, realized by two-phase commit protocols as used for instance in distributed database management systems, or by deploying newer developments such as atomic block writes [131]. The provision for one such approach would raise the MTTDL of the ClusterRAID to the regime of more than 10^8 hours.

For similar boundary conditions, i.e. comparable bandwidth and storage requirements, it has been shown that systems employing erasure-resilient codes are more reliable in terms of the MTTDL than systems simply relying on the replication of data [112]. Therefore, the next section will introduce the theory of error- and erasure correcting codes as used in the ClusterRAID.

4. Error and Erasure Correcting Codes

> *If the machine can detect an error, why can't it*
> *locate the position of the error and correct it?*
> Richard Hamming

This chapter provides a short survey of the theory of error and erasure correcting codes, as far as they are relevant for the ClusterRAID. After introducing the fundamental terms and necessary concepts, the algebra over Galois fields and their relation with polynomials will be presented as the basis for the subsequent discussion of the Bose-Chadhuri-Hocquenghem (BCH) codes. Following this, the BCH sub-class of Reed-Solomon (RS) codes and their special variant used for the ClusterRAID will be presented. A discussion of additional algorithms that might be implemented in future versions of the ClusterRAID closes this chapter. A more complete and in-depth discussion on the theory of this topic can be found in [132, 133, 134, 135, 136, 137], from which most of the introductory part has been taken.

4.1. Introduction

Communication systems are means to transport information from one place (the source) to another place (the sink). On an abstract level, these systems can be represented by the block diagram given in figure 4.1. The information source is the origin of a message M which is sent to the information sink. In order to facilitate the actual transmission over a communication channel, the message is processed by an encoder. This processing may include the mapping of information symbols onto signals which can be efficiently transmitted over the channel or the adding of redundancy information to protect against transmission errors. During the actual transport of the encoded information C over the transmission channel, a signal may be corrupted by external distortions that are usually abstracted in the term *noise*. Hence, the signal W, as received by a decoder, is a superposition of the originally encoded message C and the noise E. It is the decoder's task to extract M from W. The result of the decoding M', which is not necessarily identical to M, is then passed to the information sink.

Radio, television, cellular telephones, or communication with satellites are typical examples of such data transmission systems. However, this model also applies to storage systems: in this case the channel is the storage medium and the transmission is between two points in time, i.e. the point in time at which the data is stored and the point at which it is retrieved.

Coding theory in general concerns methods aimed at making the transfer efficient and accurate. The theory of error control coding in particular deals with the detection and correction of transmission errors, which affect data integrity. As this chapter introduces the fundamentals of error correction, the discussion will be focused on the processing steps of encoding a message M and decoding the

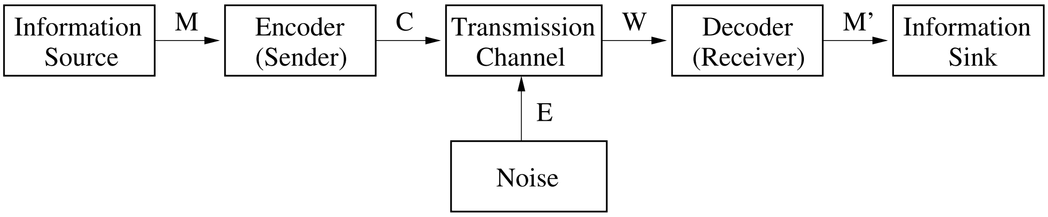

FIGURE 4.1.: Basic elements of a data transmission system.

potentially corrupted message W.

Encoding a message can be viewed as mapping a sequence of input symbols, the message, to elements of a *code alphabet*, often called *code*. In digital communication systems the symbols to be encoded and the elements of the code alphabet are usually represented in binary form. A *binary code* C is defined as a set of sequences of the digits 0 and 1. These sequences are referred to as *binary words*. The set of the two digits will be denoted by $K = \{0,1\}$, and the set of all binary words of a certain length n is K^n. If all words in a binary code have the same length it is called a *block code*. As the rest of this chapter will only deal with block codes, the term *block* will be omitted for reasons of convenience.

The sum (and the difference) of two words in K^n is defined as the bitwise sum modulo 2. A code C is *linear* if for any two words $y,z \in C$, their sum $y+z$ is also an element of C. Such codes are thus closed under addition. For instance, the code $C_1 = \{00,01\}$ is a linear code, while the code $C_2 = \{00,01,10\}$ is not. Linearity is an important characteristic of a code and all codes discussed later in this chapter will be of this type.

The *distance* $d(y,z)$ of two code words $y,z \in C$ is the number of digits, in which y and z differ. The *weight* $w(y)$ of a word $y \in C$ is the number of non-zero digits in y. The *distance of a code* C is the minimum distance of any two code words. As can be easily shown, for a linear code this minimum distance is equal to the minimum weight of any non-zero code word. For a non-linear code, however, the above statement does not hold. An example is the non-linear code $C = \{0111,1101\}$. The code distance is 2, but the minimum weight is 3.

Many techniques for studying the properties of linear codes have their origins in linear algebra. This is due to the fact that binary words of length n can be considered as elements of the vector space K^n and that linear codes form subspaces of this vector space. As usual, the number of entries in a basis for a vector space is called its *dimension*. The dimension of a linear code is the dimension of its corresponding vector space. The three characteristics of a linear code C (length n, dimension k, and distance d) are usually abbreviated in the form $C(n,k,d)$. It will become clear later that the dimension k is the length of the words to be encoded. The length n is the length of the encoded word, i.e. the number of information digits plus the number of redundancy digits, while the distance d is important for the error-correcting capability of the code.

The *dual code* $C^{\perp}$ of a linear code C is defined as $C^{\perp} = \{w : w \cdot v = 0 \mid v \in C, w \in K^n\}$. The product of two words in K^n is the usual scalar product of vectors. The *linear span*, denoted <S>, of a set of binary words $S = \{v_1, v_2, \ldots, v_m\}$ is the set of all linear combinations of vectors in S. These definitions will be valuable when dealing with the concept of generator and parity check matrices in the following paragraph.

A *generator matrix* for a linear code $C(n,k,d)$ is the matrix $G = g_{i,j}$ the rows of which form a basis for C. Instead of listing all its elements, this matrix provides a concise representation of C. Obviously, the matrix has n columns and rank k. Normally, the generator matrix is given in *row echelon form (REF)* or even in *reduced row echelon form (RREF)*.[1]
A code $C(n,k,d)$ can encode all messages $u \in K^k$. For $u \in K^k$ and a generator matrix G the encoded message is $v = u \cdot G$. As an example, consider the message $u = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$ and the generator matrix G in RREF as given below:

$$v = u \cdot G = (0\ 1\ 1) \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} = (\underbrace{0\ 1\ 1}_{u}\ 1\ 1) \tag{4.1}$$

If the generator matrix is of the form $\begin{bmatrix} I_k X \end{bmatrix}$, where I_k is the identity matrix, then G is in *standard form* and the generated code is called *systematic*. Systematic codes have the advantage to include the unmodified original message, while the redundancy information is contained in the remaining bits. This makes decoding easy, as is shown in the example above.
A matrix H is a *parity check matrix* for a linear code C if the columns of H form a basis for $C^\perp$. For such a code $C(n,k,d)$ it follows that:

$$v \cdot H = 0 \quad \forall\, v \in C \tag{4.2}$$

The construction of H can either be done by finding a basis for $C^\perp$ or by using the *parity check equations* as explained in the following sentences. Suppose the generator matrix G of a linear (n,k,d) code C is in standard form. From this it follows that G is a $k \times n$ matrix. The last $n-k$ columns of G provide the parity check equations:

$$\sum_{i=1}^{k} g_{i,k+j} a_i = a_{k+j} \quad j = 1, \ldots, n-k \tag{4.3}$$

For any valid code word $a = (a_1, a_2, \ldots, a_n)$ in C these equations hold true. H is simply the representation of the above equations in matrix form. The generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{4.4}$$

may serve as an example. The parity check equations for a code word $a = (a_1, a_2, a_3, a_4, a_5)$ are

$$a_1 + a_2 = a_4 \quad \text{and} \quad a_1 + a_3 = a_5 \tag{4.5}$$

as can be easily calculated by multiplying a and G. Therefore,

---

[1] Matrices given in REF or RREF have certain structural properties. The REF of a matrix has all its zero rows at the bottom and any leading 1 is right of all leading 1s in the rows above. A *leading 1* is a 1 that has only 0s left of it in the same row. A *leading column* is a column with a leading 1. In the RREF of a matrix the leading columns have exactly one 1.

$$H = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad a \cdot H = 0 \tag{4.6}$$

The length of C is $n = 5$ and its dimension is $k = 3$. The dimension of $C^{\perp}$ is $n - k = 2$, which is reflected in the rank of H. The columns of H form a basis of $C^{\perp}$, correspondingly the transpose H^T is a generator matrix of $C^{\perp}$. Here, another advantage of a code C given in standard form, i.e. $G = \left[I_k X \right]$ is revealed: H can be immediately constructed as $H = \left[{X \atop I_{n-k}} \right]$. The representation of a code by its parity check matrix is equivalent to the representation by its generator matrix. However, an even more concise representation can be achieved by using polynomials, as described below.
Another important class of codes are *cyclic codes*. It turns out that BCH codes and Reed-Solomon codes belong to this class. A code C is called cyclic if, for any code word $v \in C$, its cyclic shift $\pi(v)$ is also an element of C. A word $v \in K^n$ is said to be a *generator* for a cyclic linear code C if

$$C =< \{v, \pi(v), \pi^2(v), \ldots, \pi^{n-1}(v)\} > \tag{4.7}$$

where π^i denotes the ith cyclic shift of v. Cyclic codes are easily represented in terms of polynomials. A polynomial of degree n over K is the term $a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$, where the coefficients a_i are elements of K. $K[x]$ denotes the set of all polynomials over K, $K^n[x]$ the set of all polynomials of degree smaller than n. A code word v of length n corresponds to a polynomial $v(x) \in K^n[x]$ with the digits regarded as coefficients. For instance, the word 1010 represents the polynomial $1 + 0x + 1x^2 + 0x^3$. In order to simplify the identification of a code word with its polynomial, the latter are given throughout this chapter with the lowest order of x first. In terms of polynomials, the product $\pi(v) = x\,v(x) \bmod (1 + x^n)$ defines a cyclic shift.[2] The *generator polynomial* of a cyclic linear code C is the unique non-zero polynomial of minimum degree in C. Proofs for the uniqueness and the generating characteristics of this polynomial can be found in [134]. A generator matrix G can be obtained from the generator polynomial $g(v)$ and its $k-1$ cyclic shifts:

$$G = \begin{pmatrix} g(x) \\ xg(x) \\ x^2 g(x) \\ \vdots \\ x^{k-1} g(x) \end{pmatrix} \tag{4.8}$$

The generator polynomial of a cyclic linear code is the code's most concise representation. The encoding of a message $a = (a_0, a_2, \ldots, a_{k-1})$, that is $a(x) = a_0 + a_2 x + \cdots + a_{k-1} x^{k-1}$, is given by polynomial multiplication with the generator polynomial $g(x)$:

$$e(x) = (a(x) \cdot g(x)) \tag{4.9}$$

---

[2] For polynomials, $f(x) \bmod h(x)$ is the remainder of the division of $f(x)$ by $h(x)$: $f(x) \bmod h(x) = c(x)$ if there exists a function $g(x)$ with $f(x) = h(x)g(x) + c(x)$. Two polynomials $p(x), q(x)$ are said to be *equivalent mod* $h(x)$ or *congruent mod* $h(x)$, denoted by $p(x) \equiv q(x)$, if $p(x) \bmod h(x) = r = q(x) \bmod h(x)$.

code word	polynomial in x mod h(x)	power of β
000	0	-
100	1	β^0
010	x	β^1
001	x^2	β^2
101	$x^3 \equiv 1+x^2$	β^3
111	$x^4 \equiv 1+x+x^2$	β^4
110	$x^5 \equiv 1+x$	β^5
011	$x^6 \equiv x+x^2$	β^6

TABLE 4.1.: The elements of $GF(2^3)$, constructed using $h(x) = 1+x^2+x^3$.

Decoding is then done in analogy by polynomial division.

Many ideas of coding theory are based on the concept of *finite* or *Galois fields*. One aspect of such fields is illustrated during the discussion of the algorithm implemented for the ClusterRAID. They are characterized by a finite number of elements, and their relation to polynomials in $K^n[x]$ or elements in K^n will be introduced in the following.

As discussed above, every polynomial in $K^n[x]$ corresponds to an element in K^n. The addition (subtraction), multiplication, and division of polynomials modulo a function $h(x)$ thus define the corresponding operations in K^n. However, in order for K^n to obtain all properties of a field, it is not sufficient to define the multiplication modulo an arbitrary function. In general, the multiplication over K^n must be defined modulo an *irreducible polynomial* of degree n [134]. A polynomial $f(x) \in K[x]$ is said to be irreducible over K if there are no divisors of $f(x)$ except for 1 and $f(x)$ itself. If this requirement is fulfilled, however, the corresponding algebraic structure has all properties of a field with a finite number of elements, i.e. a Galois field.

An *extension field GF(p^m)* is a Galois field with p^m elements, where p is a prime and m is an integer number. As we are dealing with binary words, the Galois fields with $p = 2$ are of particular interest. The length of any given field element is m, while the number of field elements is 2^m.

The construction of a Galois field can be eased if the function $h(x)$ is *primitive*. A polynomial of degree $n > 1$ is said to be primitive if it does not divide $1+x^m$ for any $m < 2^n - 1$. Given a primitive polynomial $h(x)$ of degree n, the Galois field $GF(2^n)$ can be constructed by powers of β modulo h: $\beta^n = x^n \bmod h(x)$.

A simple example using the field $GF(2^3)$ illustrates these concepts. A primitive polynomial to define the multiplication of elements of this field is $h(x) = 1+x^2+x^3$, i.e. there is no $m < 2^n - 1 = 7$ for which $h(x)$ divides $1+x^m$. The field elements, which can be calculated as powers of β as $\beta^n = x^n \bmod h(x) = x^n \bmod (1+x^2+x^3)$, are given in table 4.1. If any non-zero element of a Galois field can be expressed by the powers of an element α, then α is said to be a primitive element. In the example above, β is such a primitive element. The elements of a field are then given by $\{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^n-2}\}$.

An element $\gamma \in GF(2^p)$ is said to be a *root* of a polynomial $f(x) \in K^n[x]$, if $f(\gamma) = 0$. The *minimal polynomial* of γ is the polynomial of smallest degree in $K[x]$ for which γ is a root. The notation of a minimal polynomial for $\gamma \in GF(2^p)$ will be m_γ, but if $GF(2^p)$ has been constructed using a primitive element α, where $\gamma = \alpha^i$, then the notation m_i will be used. These minimal polynomials will be essential when defining the BCH and RS codes.

There are two important bounds for codes, namely the *Hamming bound* and the *Singleton bound*. The Hamming bound is a general statement on the size of a code C, denoted by $|C|$, when the length of the code is n and the distance is $d \geq 2t+1$ ($t \in \mathbb{N}$):

$$|C| \leq \frac{2^n}{\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{t}} \qquad \text{(Hamming bound)} \tag{4.10}$$

This bound results from the fact that the number of words that have a distance smaller than or equal to t from a given code word is $\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{t}$. As this set of words is disjoint for any given code word of C, the total number 2^n of all words of length n is an upper bound for the product of all code words and the words in a distance smaller or equal to t. If a code C attains the bound, it is called a *perfect* code. It can be shown that Hamming codes are perfect codes.

The Singleton bound will play an important role for the RS codes. For any given linear (n,k,d) code C this bound states

$$d-1 \leq n-k \qquad \text{(Singleton bound)} \tag{4.11}$$

A proof of this inequality can be found in [134]. This bound can be viewed as limiting the distance of a code, i.e. its error-correcting capability t, if length and dimension are given. A code with distance d can correct t errors or $2t$ erasures:

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor \tag{4.12}$$

If the code has a distance $d = n-k+1$, the code is said to be *maximum distance separable (MDS)*. In the case of erasure decoding, this translates to the fact that every additional redundancy symbol, i.e. an increased n and a constant k, enables the code to tolerate an additional erasure. As will be shown later, the codes deployed in the ClusterRAID are MDS codes.

4.2. Hamming Codes

The concepts of parity check matrices and minimal polynomials allow for an elegant description of cyclic Hamming codes of length $n = 2^r - 1$, for each $r > 2$ [138]. If β is a primitive element of $GF(2^r)$, then the $(2^r-1) \times r$ parity check matrix

$$H = \begin{pmatrix} 1 \\ \beta \\ \beta^2 \\ \vdots \\ \beta^{2^r-2} \end{pmatrix} \tag{4.13}$$

defines a cyclic Hamming code, which can correct single bit-errors. The decoding scheme relies on the fact that a received word w can be written as the code word sent and an error polynomial: $w(x) = c(x) + e(x)$. The product wH reduces to eH, and for a single bit error this gives the power of β, which is the most likely error position. The product $wH = eH = s(x)$ defines the *syndrome*

polynomial or *syndrome s(x)*, which contains information about the error locations. Obviously, if $s(x) = 0$, no error could be detected. The following example illustrates the decoding procedure.
Let $GF(8)$ be constructed using the polynomial $h(x) = 1 + x^2 + x^3$ as in table 4.1. The parity check matrix for a Hamming code would be

$$H = \begin{pmatrix} 1 \\ \beta \\ \beta^2 \\ \beta^3 \\ \beta^4 \\ \beta^5 \\ \beta^6 \end{pmatrix} = \begin{pmatrix} 100 \\ 010 \\ 001 \\ 101 \\ 111 \\ 110 \\ 011 \end{pmatrix} \tag{4.14}$$

If the word $w = (1001001)$ is received, the syndrome is $w(\beta) = 100 + 101 + 011 = 010 = \beta$. Thus, the error polynomial is $e(x) = x$, which leads to $c(x) = w(x) + e(x) = 1101001$ as the most likely correct word.

4.3. BCH Codes

Bose-Chaudhuri-Hocquenghem, or *BCH codes* for short, have been described independently by R. C. Bose and D. K. Ray-Chaudhuri [139] and A. Hocquenghem [140]. These codes form an extensive class of error correcting codes with an easy decoding scheme. Reed-Solomon codes discussed in the next section are a non-binary subclass of these codes. Although BCH codes can be constructed to correct an arbitrary number of errors, the discussion will be confined to 2-error correcting BCH codes, as they provide an illustrative example.
The 2-error correcting BCH code of length $n = 2^r - 1$ is constructed using the generator polynomial $g(x) = m_\beta(x) m_{\beta^3}(x)$, where β is a primitive element in $GF(2^r)$. The dimension of this code is $k = 2^r - 2r - 1$, since $deg(g(x)) = 2r$ [134]. The $(2^r - 1) \times (2r)$ parity check matrix H is given by

$$H = \begin{pmatrix} 1 & 1 \\ \beta & \beta^3 \\ \beta^2 & \beta^6 \\ \vdots & \vdots \\ \beta^{2^r-2} & \beta^{3(2^r-2)} \end{pmatrix} \tag{4.15}$$

This matrix defines a $(2^r - 1, 2^r - 2r - 1, 5)$ code [133, 134], where the distance $d = 5$ reflects the 2-error correcting capability, see equation 4.12.
Decoding corresponds to the procedure for Hamming codes in the previous section. The syndrome $wH = eH = (e(\beta), e(\beta^3)) = (s_1, s_3)$ is used to construct the error positions. If the syndrome is 0, no error could be detected. If there has been one error during transmission, then $s_1^3 = s_3 = \beta^i = x^i$. For the case of two errors it can be easily shown that the roots of

$$x^2 + s_1 x + \left(\frac{s_3}{s_1} + s_1^2 \right) = 0 \tag{4.16}$$

provide the error locations as powers of the primitive element β [134]. This polynomial is therefore also referred to as the *error locator polynomial*.

As an example, let (01000000) be the syndrome of a received word of the BCH code for $GF(2^4)$ with elements as given in table A.1. Equation 4.16 reduces to

$$x^2+\beta x+\beta^2=0 \tag{4.17}$$

with roots β^6 and β^{11}. Thus, the error polynomial is $e(x)=x^6+x^{11}$ requiring bits 6 and 11 to be toggled to obtain the most likely correct code word.

4.4. Reed-Solomon Codes

The codes to be discussed throughout this section were discovered by I. Reed and G. Solomon in 1960 [141]. They are now very widespread, used in consumer electronics applications, such as CD-ROM, DVD, DVCR, or HDTV, as well as for space communication links by NASA and ESA. They have also been used for communication protocols, for cryptography purposes, or in peer-to-peer storage systems.

One of the reasons for the popularity of Reed-Solomon (RS) codes is that they attain the Singleton bound, i.e. they achieve the largest minimum distance between codewords for a given block length n. Thus, for these codes

$$d=n-k+1 \tag{4.18}$$

RS codes are capable of correcting $t=\lfloor\frac{n-k}{2}\rfloor$ errors. However, for RAID systems the erasure correcting capability, denoted by ρ, is of more importance:

$$\rho=d-1=n-k \tag{4.19}$$

where n is the length of a code word, while the dimension k is the length of the message words to be encoded. The difference $n-k$ is the number of redundancy symbols. Thus, for one additional redundancy symbol, RS codes are able to handle one additional erasure. It is intuitively clear that this must be a lower bound for the erasure correcting capabilities of a code: if it is able to cope with $n-k$ erasures, there must be at least $n-k+1$ references to the corresponding data.

As already mentioned, RS codes are a sub-class of BCH codes, notably a *non-binary* sub-class. The symbols dealt with are therefore no longer single bits, but binary words. The advantage of using more than one bit per symbol is that for any given n and k the ratio of codewords to the number of all vectors in the whole vector space is decreased. This allows for a larger distance d compared to binary codes and thus for larger t and ρ.

RS codes are defined as roots of polynomials, similar to BCH codes in the former section. For $GF(2^r)$ and its primitive element β the generator polynomial of an RS code is given by

$$g(x)=\prod_{i=l+1}^{i=l+\delta-1}(\beta^i+x) \tag{4.20}$$

The parameter l is an arbitrary integer usually set to -1. This code has a distance $d(C)=\delta$. The code length is 2^r-1. The number of message symbols in a code word, i.e. its dimension, is $k=2^r-\delta$.

The size of the code $|C|$ is 2^{rk}, as any of the k symbols in a message can be any of the 2^r elements of $GF(2^r)$. Usually, RS codes are denoted by $RS(2^r, \delta)$.
To encode a message

$$m(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{k-1} x^{k-1} \tag{4.21}$$

the code word is again obtained by polynomial multiplication

$$c(x) = m(x)g(x) = \sum_{i=0}^{n} c_i x^i \tag{4.22}$$

As an example, an RS(8,3) code will be constructed using $1 + x^2 + x^3$ and $GF(2^3)$. Let $g(x) = (1 + x)(\beta + x)$ generate a code C. The code has length $n = 2^r - 1 = 7$. Since $r = 3$ and $k = n - deg(g(x)) = 5$, C has $2^{rk} = 2^{15}$ code words. Using table 4.1, $g(x)$ can be expressed as

$$\begin{aligned} g(x) &= (1+x)(\beta+x) && (4.23)\\ &= \beta + \beta x + x + x^2 && (4.24)\\ &= \beta + (1+\beta)x + x^2 && (4.25)\\ &= \beta + \beta^5 x + x^2 && (4.26) \end{aligned}$$

To encode the message $m = 1\beta^4 000$, $m(x)$ is multiplied by $g(x)$:

$$c(x) = m(x)g(x) = (1 + \beta^4 x)(\beta + \beta^5 x + x^2) = \beta + \beta^3 x^2 + \beta^4 x^3 \tag{4.27}$$

The transmitted code word would be $\beta 0 \beta^3 \beta^4 000$.
In order to generate a systematic code, the message polynomial $m(x)$ can be multiplied by $x^{\delta-1}$ and divided by $g(x)$. This shifts the coefficients of the message polynomial into the upper symbols of the code word. The code word polynomial itself is defined by

$$c(x) = x^{\delta-1} m(x) - r(x) \tag{4.28}$$

where $r(x)$ is the remainder in

$$x^{\delta-1} m(x) = q(x)g(x) + r(x) \tag{4.29}$$

The coefficients of the polynomial $c(x)$ are the symbols of the code word.[3]
The decoding of non-binary codes, such as RS codes, is lightly more intricate than for binary codes. In addition to finding the error locations, the sizes of the errors must also be determined. For an RS code C over $GF(2^r)$ constructed using a generator polynomial $g(x)$ as given in equation 4.20, let $a_1, a_2, \ldots, a_e$ be the locations of the errors and $b_1, b_2, \ldots, b_e$ their corresponding magnitudes for a received word w. The syndromes s_j required to determine the most probable code word are given by

$$s_j = w(\beta^j) = c(\beta^j) + e(\beta^j) = e(\beta^j) = \sum_{i=1}^{t} b_i a_i^j \quad \text{, where } t = \lfloor(\delta - 1)/2\rfloor \geq e \tag{4.30}$$

---

[3] This is known as the BCH view of Reed-Solomon codes. In the original paper, Reed and Solomon calculated the code word symbols as the values of a polynomial, the coefficients of which were the message symbols.

If $e < t$, the a_i for $e+1 \leq i \leq t$ are defined as zero for reasons of convenience. Since $m+1 \leq j \leq m+\delta-1$, there are $\delta - 1$ equations for the $2e$ unknowns a_j and b_j. The error locator polynomial

$$\sigma_A(x) = (a_1 + x)(a_2 + x) \cdots (a_e + x) = x^e + \sum_{i=0}^{e-1} \sigma_i x^i \tag{4.31}$$

has the error positions as its roots. The syndromes s_j satisfy the linear recursion [142, 143]

$$s_{j+e} = \sum_{i=0}^{e-1} s_{j+i}\sigma_i \tag{4.32}$$

By substituting $j = m+1, \ldots, m+e$ this relation leads to a set of linear equations in σ_j, which can be expressed in matrix form:

$$\begin{pmatrix} s_{m+1} & s_{m+2} & \cdots & s_{m+e} \\ s_{m+2} & s_{m+3} & \cdots & s_{m+e+1} \\ \vdots & \vdots & & \vdots \\ s_{m+e} & s_{m+e+1} & \cdots & s_{m+2e-1} \end{pmatrix} \begin{pmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{e-1} \end{pmatrix} = \begin{pmatrix} s_{m+e+1} \\ s_{m+e+2} \\ \vdots \\ s_{m+2e} \end{pmatrix} \tag{4.33}$$

Since the syndromes are known from equation 4.30, the roots of the error locator polynomial can be calculated. These roots are the positions of the errors a_j. This allows equation 4.30 to be solved for the magnitudes of the errors b_j

$$\begin{pmatrix} a_1^{m+1} & a_2^{m+1} & \cdots & a_e^{m+1} \\ a_1^{m+2} & a_2^{m+2} & \cdots & a_e^{m+2} \\ \vdots & \vdots & & \vdots \\ a_1^{m+e} & a_2^{m+e+} & \cdots & a_e^{m+e} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_e \end{pmatrix} = \begin{pmatrix} s_{m+1} \\ s_{m+2} \\ \vdots \\ s_{m+e} \end{pmatrix} \tag{4.34}$$

As an example, consider the $RS(8,5)$ code over $GF(2^3)$ constructed using $h(x) = 1 + x^2 + x^3$, see table 4.1. The primitive element of the code is denoted by β and its generator is given by

$$g(x) = (1+x)(\beta + x)(\beta^2 + x)(\beta^3 + x) = \beta^6 + \beta^4 x + \beta x^2 + \beta x^3 + x^4 \tag{4.35}$$

A received code word may be $w = 00\beta\beta100$. The syndromes s_j for $1 \leq j \leq \delta - 1$ are

$$\begin{aligned} s_0 &= w(\beta^0) = \beta + \beta + 1 = 1, && (4.36) \\ s_1 &= w(\beta^1) = \beta^3 + \beta^4 + \beta^4 = \beta^3, && (4.37) \\ s_2 &= w(\beta^2) = \beta^5 + \beta^7 + \beta^8 = 0 \text{ , and} && (4.38) \\ s_3 &= w(\beta^3) = \beta^7 + \beta^{10} + \beta^{12} = \beta^4 && (4.39) \end{aligned}$$

Taking the syndromes into account, the linear system in equation 4.33 can be solved for σ_0 and σ_1. The error locator polynomial σ_A is then given by

$$\sigma_A(x) = \beta + \beta^5 x + x^2 = (1+x)(\beta + x) \tag{4.40}$$

The most probable locations of errors are therefore β^0 and β^1. With these values the linear system 4.34 can now be solved

$$\begin{pmatrix} 1 & 1 \\ \beta^0 & \beta^1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 1 \\ \beta^3 \end{pmatrix} \leftrightarrow \begin{pmatrix} 1 & 1 \\ 0 & \beta^5 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 1 \\ \beta^2 \end{pmatrix} \tag{4.41}$$

Thus, $b_2 = \beta^4$ and $b_1 = \beta^6$. The most likely code word is therefore

$$c = w + e = 00\beta\beta 100 + \beta^6\beta^4 00000 = \beta^6\beta^4\beta\beta 100 \tag{4.42}$$

RS codes are also capable of decoding transmitted words with both errors and erasures [134]. For systems in which all the error locations a_j are known, the syndromes 4.30 can be immediately solved for the error magnitudes b_j. RAID-like systems are an example of such a scenario.

More efficient algorithms than the presented ones have been developed in order to speed up the error and erasure decoding of RS codes, including for instance the Berlekamp-Massey algorithm [144, 145] or Euclid's algorithm [136]. Optimizing the algorithms is still an active field of research [146, 147]. However, instead of discussing any of the alternative approaches in detail, the erasure-resilient RS variant used in the ClusterRAID system will be now presented.

4.5. Vandermonde-based Reed-Solomon Codes

The form of Reed-Solomon codes given in the previous section is elegant and appropriate from a coding theorist's point of view. However, it is not very well suited for implementation in a programming language. Furthermore, for RAID-like systems the capability of dealing with erasures is more relevant than the classic field of error detection and correction. Although RS codes can also cope with erasures, a special variant of RS codes has been developed that is more adapted to this field of application [148, 149]. The ClusterRAID prototype is based on this version of RS codes, i.e. the *Vandermonde-based Reed-Solomon Codes*.

The basic idea is to define a $(k+\delta-1) \times k$ generator matrix F' such that a quadratic submatrix of F', formed by the deletion of any $\delta - 1$ rows, is still invertible. The construction of F' starts from the Vandermonde matrix $F = f_{i,j}$, where $f_{i,j} = i^j$:

$$F = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \dots & 0^{k-1} \\ 1^0 & 1^1 & 1^2 & \dots & 1^{k-1} \\ 2^0 & 2^1 & 2^2 & \dots & 2^{k-1} \\ \vdots & \vdots & \vdots & & \vdots \\ (k+\delta-2)^0 & (k+\delta-2)^1 & (k+\delta-2)^2 & \dots & (k+\delta-2)^{k-1} \end{pmatrix} \tag{4.43}$$

Since the rows of the matrix F are linearly independent, F is invertible. Primitive transformations, i.e. the multiplication of a row with a constant, the swapping of rows, or the addition of two rows, do not change this characteristic of F. Using these primitive operations, F can be transformed into a matrix F', such that the first k rows form the identity matrix

$$F' = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ f'_{k,0} & f'_{k,1} & f'_{k,2} & \cdots & f'_{k,k-1} \\ \vdots & \vdots & \vdots & & \vdots \\ f'_{k+\delta-2,0} & f'_{k+\delta-2,1} & f'_{k+\delta-2,2} & \cdots & f'_{k+\delta-2,k-1} \end{pmatrix} \tag{4.44}$$

This matrix F' is the generator matrix of a systematic RS code. The encoding of a message $m = (m_0, m_1, \dots, m_{k-1})$ is done by multiplication with F':

$$F' \begin{pmatrix} m_0 \\ m_1 \\ \vdots \\ m_{k-1} \end{pmatrix} = \begin{pmatrix} m_0 \\ m_1 \\ \vdots \\ m_{k-1} \\ c_0 \\ c_1 \\ \vdots \\ c_{\delta-2} \end{pmatrix} \tag{4.45}$$

with check values c_0 through $c_{\delta-2}$. The message symbols m_i and the check values c_i can be directly associated with devices they are stored on, i.e. k is the number of data devices and $\delta - 1$ is the number of redundancy or check devices. The number of rows thus corresponds to the number of devices in the system. In case of failures, the rows in F' that represent the failed devices are removed from the matrix. As mentioned above, the matrix continues to be invertible even if $\delta - 1$ rows are removed. Thus, the linear equation system given in equation 4.45 can be solved for the message symbols m_i even if up to $\delta - 1$ rows (devices) are lost. From the reconstructed data any information stored on possibly failed redundancy devices can be reconstructed. The solution of equation 4.45 requires infinite precision, otherwise rounding errors could compromise the results. If, however, the coefficients of F' are chosen from a Galois field $GF(2^r)$ and the arithmetic is performed over this field, the precision problem does not arise. Thus, the matrix elements of F' are elements of a Galois field, and so are the symbols of a message and the check words. As a restriction, the Galois field must be chosen such that the size of the field 2^r is larger than $k + \delta - 1$ [132]. For instance, 16-bit-wide symbols, i.e. $r = 16$, allow for up to 65,535 devices, a value rarely reached in practice.

As the relation between message symbols and check words is linear, the necessary update of a check word from c_i to c'_i after a message symbol's update can be determined from the difference between the new and the old message symbol. Consider the update of the message symbol m_j to m'_j

$$\triangle c_i = c'_i - c_i = f'_{i,j}(m'_j - m_j) \tag{4.46}$$

Aside from the message symbol to be changed, no other message symbol need to be considered when calculating the contribution of the changed message symbol to the encoding information. The update penalty, i.e. the number of devices to be updated per data device update, is minimal for these codes.

In all RAID systems updates of data blocks always imply the update of the block holding the corresponding redundancy information. This information can be calculated immediately if all data associated with the block concerned is at hand. For instance, if the amount of data written to a RAID-5 system is larger than one stripe, the parity chunk can be determined directly using the XOR operation. No read access to the underlying devices is necessary. However, for small writes there are two possibilities. Either the case is attributed to the one of large writes by reading the missing data from the other devices, or the difference of the data on the device and the data to be written is used to determine the difference between the old and new parity information. The latter approach includes four I/O operations for one small write. It is therefore chosen by most RAID implementations, since the former generates even more than four I/O operations in configurations with at least five disks. Despite this, the response time of small writes is deteriorated approximately by a factor of two.[4] Since the ClusterRAID does not stripe data over several devices and since any read access to other data devices includes network transactions, every write access implies several steps. Old data must be read from the device, the difference has to be determined and is then used to calculate the correction of the information on the redundancy blocks residing on a remote node. The calculation is performed using the Vandermonde-based RS codes as described above. The correction of the redundancy blocks is done by applying the XOR operation on the old redundancy information and its update. This also implies two I/O operations per redundancy block.

A simple approach to determine the complexity reduces the process of updating data in the ClusterRAID to the following operations and their respective durations: read a data block from disk (T_R), write a data block to disk (T_W), perform an XOR operation on two symbols (T_X), and perform a Galois field multiplication (T_M). These operations are all done on a block of size S symbols. The sequence of updating one or several symbols in a block is then: read the block from disk, XOR it with the new data, write out the new data, apply the redundancy algorithm on the difference, and update the redundancy data on the check disks, again using the XOR operation. The cost of a block update in terms of number of operations, denoted by c_{op}, is therefore:

$$c_{op} = S(c_R + (\delta - 1)c_M + \delta(c_X + c_W)), \tag{4.47}$$

where the indices (R, M, X, and W) denote the same operations as above. Some of these operations can be carried out in parallel. The calculation of the redundancy data may be parallelized if the encoders reside on different nodes or are performed by customized hardware. Furthermore, the writing of the data to the check disks can be done in parallel. Therefore, the total execution time of a single write to complete on a symbol basis would be

$$T_{total} = T_R + 2T_X + max(T_M + T_R) + T_W \tag{4.48}$$

Further interleaving is possible if the read of the check data can be issued at the same time as for the local data.

---

[4] There are approaches to overcome the overhead introduced by this well-known problem of RAID5, such as caching, floating parity, or parity logging [110].

4.6. Other Algorithms

Although the Vandermonde-based Reed-Solomon codes, as set out in the previous section, are well suited for RAID-like systems and provide a number of advantageous features, the algebra involved must be performed over a Galois field. Hence, operations such as encoding and decoding are computationally intensive compared to simpler approaches such as parity calculation. Therefore, it may be worth considering alternative erasure-correcting algorithms that are less computationally expensive, but provide comparable features.

Two candidates for alternative codes are the EvenOdd scheme and the Tornado Codes. The EvenOdd scheme [150, 151] only uses XOR operations for the calculation of redundancy information. It avoids the more complex operations over finite fields required for the Reed-Solomon codes and therefore reduces the load on the host processor. Moreover, it can be easily mapped onto existing parity hardware, such as RAID controllers. Compared to simple parity approaches, EvenOdd only requires twice as many XOR operations. This is optimal, since two redundancy devices instead of just one are used. Compared to Reed-Solomon codes, the number of XOR operations is reduced by a a factor of two. The disadvantage of this approach, however, is that the algorithm is limited to correct at most two erasures. For larger systems, this may not be sufficient. In addition, the update penalty is not minimal for all data symbols. Upon change of specific data symbols the algorithm requires the update of more than just the minimal two redundancy symbols. This may have a negative impact on the performance.

Tornado codes [152, 153] form a class of recently developed erasure-correcting codes. Derived from Gallager codes [154], they are based on bipartite, irregular graphs. They are almost as space-efficient as Reed-Solomon codes, and have a linear encoding and decoding complexity. Performance evaluations indicate that Tornado codes can provide a superior en- and decoding performance for large block lengths and a large number of redundancy symbols [155]. For a small number of redundancy symbols, as required in RAID-like systems, a comparison, however, has not yet been performed. Furthermore, these codes are protected by patents.

5. ClusterRAID Implementation

This is how we did it.
Linus Torvalds

This chapter presents the prototype implementation of the ClusterRAID architecture by a set of kernel modules for the 2.4 series of the GNU/Linux operating system. Since excellent in-depth references to the Linux kernel [114, 156] exist, and hereby also to block devices, a detailed discussion of block device specific kernel internals is omitted. Only the fundamental concepts essential for the understanding of the ClusterRAID are presented at the beginning, in a section covering the interplay of the ClusterRAID driver with the core kernel. After the following architectural overview, the actual implementation is introduced from a functional approach: going from simple to more complex operational situations, the concepts used in the prototype implementation are discussed as they are necessary in the corresponding context. Of course, the description of the prototype implementation cannot be on a line by line basis. Rather, the basic data structures, algorithms, and concepts are set out. Additional functionalities, features, and further explanations are covered thereafter in a separate section, in order to ensure a concise description of the data paths.

5.1. Kernel Interfacing

From a functional point of view, the architectural considerations in section 3.2 define the ClusterRAID as being a part of the operating system. Consequently, the prototype implementation presented in this chapter is based on Linux kernel modules, i.e. dynamically loadable extensions of the GNU/Linux operating system. If not otherwise stated, the term *ClusterRAID driver* is used throughout this whole chapter to refer to this set of kernel modules.

The Linux kernel is a complex object in itself. By this time of writing the current kernel of the 2.4 series comprises about 4,500,000 lines of code, distributed over more than 10,000 files. In order to avoid unwanted side-effects, it is in general desirable to reduce the interaction of modules with the kernel to a minimum [157]. Of course, it is impossible for a module to avoid all communication. It must be registered and initialized, it may have to allocate memory dynamically, it usually provides a control interface and must respond to corresponding calls, it must serve the requests the kernel issues to it, and upon unloading it must unregister and clean up. However, it is possible to minimize the degree of interplay between a module and the kernel during these tasks. For instance, instead of requesting memory from the kernel repeatedly, the module can manage its own local cache for frequently used data structures. On this account, the prototype implementation's requirements for interaction with the core kernel have been restricted to a few functionalities, as outlined later in this section.

From the user interface point of view the ClusterRAID behaves like a disk, i.e. it has a block device interface. Hence, the prototype essentially consists of a block device driver – a kernel module that

```
struct buffer_head {
    ...
    unsigned long b_blocknr;        /* block number */
    unsigned short b_size;          /* block size */

    ...
    kdev_t b_dev;                   /* device (B_FREE = free) */

    ...
    kdev_t b_rdev;                  /* Real device */
    unsigned long b_state;          /* buffer state bitmap (see above) */

    ...
    char * b_data;                  /* pointer to data block */
    struct page *b_page;            /* the page this bh is mapped to */
    void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
    void *b_private;                /* reserved for b_end_io */

    unsigned long b_rsector;        /* Real buffer location on disk */

    ...
};
```

FIGURE 5.1.: The kernel's structure `buffer_head`. The above code is directly copied from the `buffer_head` definition in the Linux kernel, but only the fields relevant for the prototype implementation are shown. Omissions are indicated by three dots.

responds to the well-defined block device programming interface. In kernel terminology, *blocks* are the fixed-size sets of adjacent bytes that a block device driver transfers in a single I/O operation. The basic unit of data transfer for the controlled hardware, however, is limited by the device's own *sector size*. In Linux, the block size must be a multiple of the sector size, for hard drives typically 512 bytes, and cannot be larger than a page frame, which is 4 kilobytes. Therefore, block device drivers can support block sizes of 512, 1024, 2048, and 4096 bytes. A block's content is stored in the so-called *buffer*, an area in the main memory. The kernel stores meta information about buffers in structures of type `buffer_head`. The prototype's interaction with the core kernel is heavily based on using this structure. A `buffer_head` contains more than twenty fields, of which only the ones relevant for and manipulated by the ClusterRAID are listed in figure 5.1.

The field `b_blocknr` describes the logical location or index of this block on the device. To account for the above-mentioned distinction between block and sector, `b_rsector` holds the first sector number of this block on the device. As its name indicates, the field `b_size` is the block size. For virtual devices, i.e. devices that control no real hardware, such as the RAID or LVM drivers, the *real* device that the request is eventually for may differ from the requested device. In order to keep track of which device the buffer head originated from, two different fields, namely `b_dev` and `b_rdev`, of type `kdev_t` have been introduced. Since the ClusterRAID also is a virtual device in this sense, these fields are used in the prototype to redirect the request to the appropriate device, as will be shown later. A buffer's state is stored in the field `b_state`. This status field indicates, for instance, whether the buffer contains valid data, and it also holds information about whether the block's content is synchronized with the corresponding data on the device. The actual buffer, i.e. the memory area containing the block's data, is pointed to by `b_data`. The reference to the corresponding page structure can be found in `b_page`. The function pointer `b_end_io` contains the address of a callback function, which is used to signal to the kernel that the request of this buffer head has been serviced. As will be shown later, this callback mechanism is the interception point at which the ClusterRAID can check if a request succeeded or failed. The actual status of the request after being served by the device is the last parameter that this callback function is invoked with (`uptodate`). Depending on its value, the ClusterRAID signals the request to be finished or initiates the appropriate reconstruction action. The `b_private` field is free for use by a driver. In the ClusterRAID driver this field is used to store the original callback function (if it replaces the function in `b_end_io` with its own version) or to hold data structures to transfer information along with a buffer head through a hierarchy of function calls.

The kernel creates block device requests by means of the function `ll_rw_block()`. This function receives, among other parameters specifying the request, a list of buffer heads describing the blocks to be transferred. After performing some buffer head management actions, it invokes `generic_make_request()` to post the request to the low-level driver. This function identifies the request queue of the device the request is intended for and calls the function pointed to by the field `make_request_fn` in the queue's descriptor. This call places the request on the driver's queue. Most drivers do not need their own implementation of this function and can therefore use the kernel's `__make_request()` function. The request queue contains all requests the kernel awaits to be serviced from the driver's devices. The actual low-level I/O is then performed by the driver's request function `request_fn()`, which is also referenced through a field in the request queue's descriptor. However, like other virtual device drivers, the presented prototype implementation does not manage its own request queue. Instead, it installs its own `make_request_fn` in the drivers queue descriptor, which performs the ClusterRAID specific actions before forwarding a request to its constituent devices. Hence, this function, namely `cr_make_request()`, is the kernel's entry point to the driver when it has a request to be served. The ClusterRAID itself uses the above-mentioned `generic_make_request()` function to create the requests for the underlying devices and the callback mechanism to intercept the requests on their way back to the requester. It signals the end of a requested I/O operation by calling the original `b_end_io()` function of the buffer head that the `cr_make_request()` function was invoked with.

Buffer heads are thus the central structure by which the ClusterRAID interacts with the kernel. Since buffer heads are involved in all block I/O operations, the `buffer_head` structures are frequently allocated and released, not only by the ClusterRAID system. For such frequently requested kernel structures the kernel has so-called slab or lookaside caches, one of which is used for structures of the type `buffer_head`. Instead of allocating the needed memory by itself, this cache, namely `bh_cachep`, is used by the ClusterRAID if it needs buffer head structures or wants to release them. As will be discussed later, the ClusterRAID implementation prefers to maintain its own private cache for the actual data *buffers*, pointed to by the field `b_data` inside a structure `buffer_head`.

The remaining interaction with the core kernel is restricted to supporting data structures and functionality, such as

- registration and initialization functions during startup,
- structures and functions for list handling,
- locks and atomic operations,

- kernel threads, wait queues, and associated functions,
- ioctls for driver control,
- print functions for debugging,
- some additional calls for memory allocation,
- some additional calls for buffer head requests, and
- unregister and cleanup functions upon unloading.

Most of this functionality, however, is regarded to be non-critical, since it is either seldomly executed, such as registering or generating kernel threads, or it requires no additional resources from the kernel, e.g. list handling, which is simple pointer arithmetic. Some of the requested functionality, however, could be reduced even further in future improvements of the ClusterRAID, if these actions turn out to be system critical. For instance, one approach would be to eliminate all memory allocation and serve all memory requests from private caches. Some of these additional kernel interactions listed above will be discussed in more detail during the functional analysis in section 5.3.

5.2. Overview

From the local point of view, the ClusterRAID is an additional, intermediary operating system layer between a user application and its data on an underlying block device. This node perspective is depicted in figure 5.2. If the kernel cannot serve a user request using one of its disk caches, more precisely the buffer cache, the request is forwarded to the ClusterRAID driver. The ClusterRAID driver has at least one constituent local data device, from which the request is finally served. A detailed discussion of the request handling for the different request scenarios is given in the functional analysis of section 5.3. Some of the request types require some data exchange with remote devices. For this, the ClusterRAID driver uses the network block devices introduced in section 2.4.1. This interface is used for updates of redundancy information as well as remote data import in case a local data reconstruction is required.
Globally, the ClusterRAID is a loosely coupled set of nodes. As outlined in chapter 3, the architecture allows nodes to store data, redundancy information, or both. All data nodes use their local device for data storage exclusively and asynchronously from other nodes in the ClusterRAID system. The redundancy storage areas, however, are always shared by multiple nodes. Due to its proof of concept character the present implementation uses the architecture with dedicated redundancy nodes. Hence, a node either stores data or redundancy information on its local ClusterRAID device. A top level overview of a ten node embodiment of the prototype implementation is given in figure 5.3. In this example, eight out of ten nodes are data nodes, while the remaining two nodes store the corresponding redundancy information. The interface between the nodes is formed by an NBD backbone, i.e. a set of peer-to-peer NBD connections which masquerade remote devices as local ones. Of course, the prototype is not restricted to this specific total number of nodes nor to this number of redundancy nodes. It can deploy any combination of these, limited only by the redundancy algorithm presented in section 5.2.2.

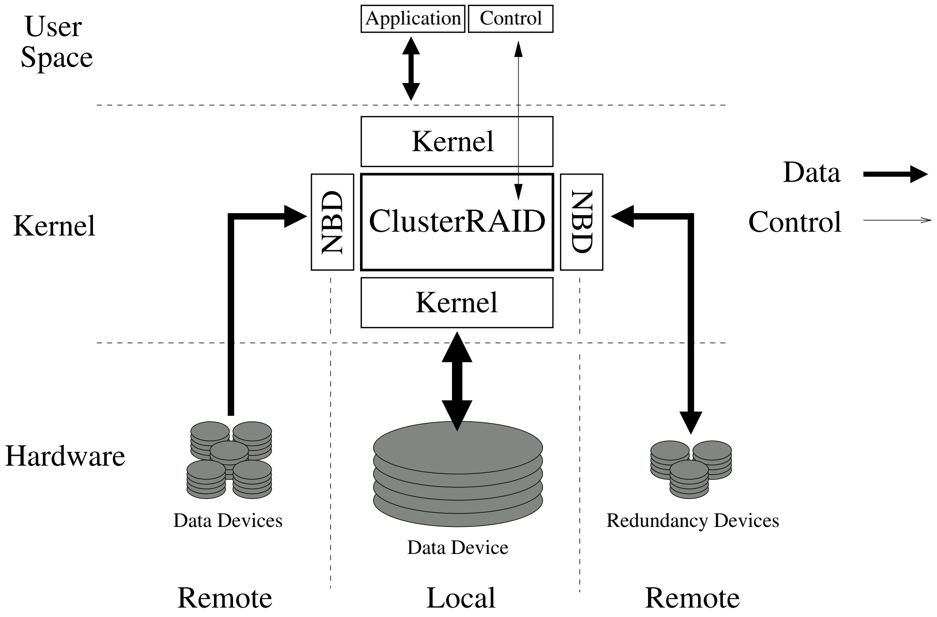
User Space
Application
Control
Kernel
Data
Control
NBD
ClusterRAID
NBD
Kernel
Hardware
Data Devices
Data Device
Redundancy Devices
Remote
Local
Remote

FIGURE 5.2.: Schematic local view of a ClusterRAID node. The ClusterRAID driver forms an additional supervision and control layer between the user application and the underlying data device.

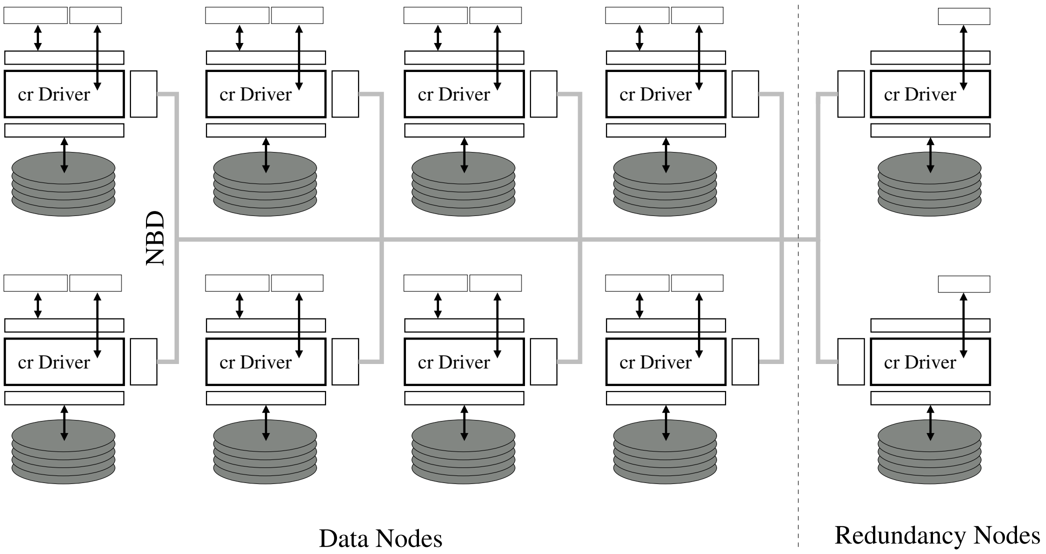
cr Driver
NBD
Data Nodes
Redundancy Nodes

FIGURE 5.3.: Schematic top level architecture of an embodiment for the prototype implementation. In this ten node example, eight nodes store data, while the remaining two store redundancy information. The interface between the nodes is a peer-to-peer NBD backbone. The unlabelled boxes correspond to the boxes in figure 5.2.

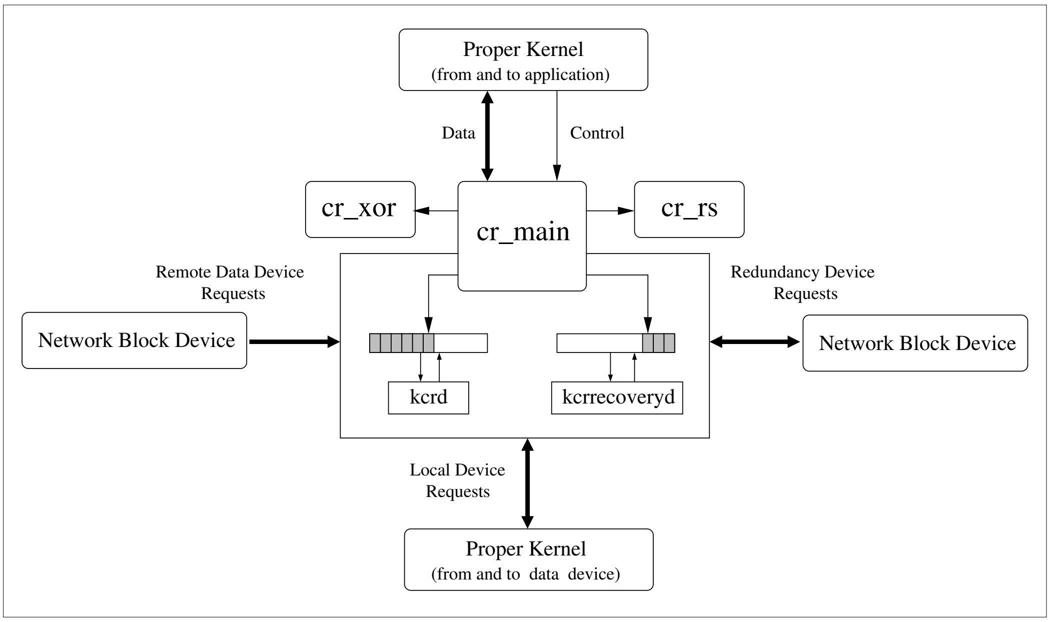
Proper Kernel
(from and to application)
Data
Control
cr_xor
cr_main
cr_rs
Remote Data Device
Requests
Redundancy Device
Requests
Network Block Device
Network Block Device
kcrd
kcrrecoveryd
Local Device
Requests
Proper Kernel
(from and to data device)

FIGURE 5.4.: Overview of the kernel level ClusterRAID software architecture.

Figure 5.4 is an overview of the prototype's software architecture. The system comprises the following main parts:

- the main module (`cr_main`),
- the codec module (`cr_rs`),
- the xor module (`cr_xor`),
- the `kcrd` kernel daemon,
- the recovery kernel daemons (`kcrrecoveryd`),
- the control program (`mkcr`, not shown in the figure), and
- the configure and setup script (`cr2launch`, not shown in the figure).

Since the modules and the kernel threads make up the central part of the ClusterRAID system, a brief overview about their structure and functional responsibilities will be given in the next subsections.

5.2.1. The Main Module

The main module `cr_main` is the center of the ClusterRAID. It provides the interface for the core kernel and, as such, is responsible for the ClusterRAID request management. The main driver combines the different required functionalities for both the data side and the redundancy side in one module. In addition to several redundancy algorithms that can be registered with the module, it is also able to administrate multiple ClusterRAID devices. The configuration of an individual device determines its type, i.e. if it is used for data or redundancy information, and thereby also the corresponding data path. Although the prototype is based on the approach of dedicated redundancy nodes, the integration of data and redundancy device functionality in a single module allows for later enhancement by other distribution schemes, as discussed in section 3.2. Along with the individual ClusterRAID data devices, references to their associated remote devices, data and redundancy, are maintained. The main module also controls all ClusterRAID kernel threads, in particular, the kernel ClusterRAID daemon `kcrd` and the set of kernel ClusterRAID recovery daemons, notably `kcrrecoveryd`. The task of data recovery is assigned to a plurality of daemons, a design choice which separates the gathering of blocks required for reconstruction from the actual decoding step. Aside from request interleaving, the reason for the use of kernel threads by the ClusterRAID is that the architecture is based on asynchronous request servicing using callback functions invoked in interrupt context. Requests that reach the driver always require data exchange with underlying devices, which can be directly attached or remote. Instead of waiting for the requests to be finished, and thus serializing the access, the requests are issued asynchronously to the kernel for treatment by the underlying devices. Once the devices have finished a request, the kernel notifies the ClusterRAID driver, via a callback function, of the completion and its status. In the time between the issue of a request and its return, the ClusterRAID driver is thus not blocked, but is available to serve new requests. Since the underlying devices notify the kernel of request completion by interrupts, the callback functions are also executed in an interrupt context. During these interrupts other interrupts are disabled. Hence, it is not allowed to call any other kernel functionality that possibly requires the caller to wait, since it cannot be reactivated, as this would also be done via interrupts. However, in the callback functions the kernel reports on a request's completion status. In some situations it may therefore be necessary to perform substantial operations at this point. For instance, upon a device failure the reconstruction of requested but currently not available data has to be initiated. Since interrupt servicing should be kept as short as possible in any case, and since furthermore the mentioned restriction limits the functionality available in callback functions, more complex operations have been transferred to the ClusterRAID kernel threads and their corresponding functionality outside the interrupt context. As can be seen from the kernel level overview in figure 5.4, the main module also uses the functionality of the two other modules of the prototype implementation, notably the `cr_xor` and the `cr_rs` module.

The data structures used in the main module and the different function classes together with representative sample methods will be introduced in the upcoming paragraphs.

Internal Data Structures

The ClusterRAID driver describes the individual devices by structures of type `cr_device_t`, as depicted in figure 5.5. Instances of this structure type contain all fundamental information required to serve ClusterRAID requests. In the ClusterRAID system the structure also identifies the device by the system wide unique value of the field `ID`. A ClusterRAID device structure also holds a field with its minor device ID, namely `__minor`, in order to be able to distinguish several devices maintained by one driver. Depending on the device type, i.e. data or redundancy, a corresponding disk structure of type `cr_disk_t`, depicted in figure 5.6, can be referenced by `data_disk` and `redundancy_disk`, respectively. These fields describe the local constituent devices. In general, an instance of the type `cr_disk_t` stores all information about the single constituent devices, the directly attached ones

```
struct cr_device_s {

     cr_disk_t         *data_disk;        // data only mode
     cr_disk_t         *redundancy_disk;  // redundancy only mode

     wait_queue_head_t wqueue;            // put processes to sleep

     struct list_head  disks;             // list of redundancy devices
     struct list_head  rdisks;            // list of remote data devices

     cr_handle_t       handles[4];        // codec handles, restricted to 4

     atomic_t          flags;
     atomic_t          faulty_devices;    // no. of failed devices
     ...
     atomic_t          redundancy_level;

     ...
     uint32_t          ID;                // unique cr device number
     ...

     int               __minor;
     int               algorithm;         // -1 means no algorithm

     kdev_t            dev;
     kdev_t            data_dev;
     kdev_t            redundancy_dev;    // redundancy only mode
     ...
};
typedef struct cr_device_s cr_device_t;
```

FIGURE 5.5.: The ClusterRAID device type. Only the most relevant fields for the operation of the ClusterRAID as such are shown. Omissions are indicated by three dots.

as well as the remote ones. All underlying remote devices are organized in lists: `disks` for the redundancy devices and `rdisks` for the remote data devices. The actual number of faulty devices is stored in `faulty_devices`. This number can be used, for instance, to decide if a write will meet the `redundancy_level`, i.e. the user-defined minimum number of redundant blocks that can be stored on the available devices. Whenever necessary, the ClusterRAID driver puts processes to sleep on the device internal wait queue `wqueue`, from which they are woken up asynchronously if data has arrived or resources have become free again.

As will be set out in the next subsection concerning the codec module, codec instances are referenced by handles. A ClusterRAID device supports up to four handles and thus four codec instances, which can be accessed simultaneously. The speed of the reconstruction in particular can benefit from the concurrent use of different codec handles by multiple kernel threads. Besides the algorithm to be used with the current device (stored in `algorithm`) and some device flags (stored in `flags`), the kernel device identifiers for this ClusterRAID device, for its data device (if any), and for its redundancy device (if any) are stored in the fields `dev`, `data_dev`, and `redundancy_dev`, respectively.

The remaining structures used by the main module for request handling all concern the management of buffer heads. Basically, these are two different classes of structures: one to serve as *meta* buffer heads, the other serves as a container for buffer heads. The former is exemplified by the type

`cr_bh_t` listed in figure 5.7.
This buffer head meta structure is an extension of the kernel's structure `buffer_head`. ClusterRAID requests, in particular write and reconstruction requests, require requests to be issued to a plurality of devices. For instance, a write request requires the storage of the generated redundancy information in addition to the real data. Similarly, for the reconstruction of data of a faulty device the corresponding redundancy information and the data of remote devices must be read. The structure contains all superordinate information about the requests, the original request to the ClusterRAID as well as all information about subsequent requests to affected devices.
A representative of the other type, `cr_bh_container_s` is listed in figure 5.8. It is mainly used to carry information about a request, which is needed in the later processing. For instance, in its `type` and `rw` fields, the driver can determine during the reconstruction of data if the original request was for a data or a redundancy device, and if data must be read or written. The ClusterRAID driver implements different flavours of this container structure, which are adapted to the specific context.

Function Classes

The internal functions of the main module can be coarsely classified into the following categories:

- core kernel interfacing,
- request handling,
- callbacks,
- thread handling,
- hashes and (remote) locking,
- device controlling and debugging,
- hooks for external modules, and
- additional helper functions.

Not all functions, however, can be unambiguously assigned to one of these function categories. For instance, the aforementioned `cr_make_requests()` function, which is the ClusterRAID's replacement function for request queue handling, can be regarded either as part of the request handling functionality or as part of the kernel interface. Similarly, callbacks are strictly speaking the last part of request handling. Some of these categories will be covered in separate sections below. The hooks for external modules are introduced in section 5.2.2, all locking and hash functionality is discussed in section 5.4.1, and controlling, testing, and debugging functionality is dealt with in section 5.5. Helper functions, such as the encapsulation of data structure allocation, are set out whenever necessary in the corresponding context.
Upon initialization, block devices register with the core kernel by calling `register_blkdev()` with a reference to a filled structure `block_device_operations`. This structure defines all operations linked to the corresponding device file. As an example, this structure holds pointers to functions which are invoked whenever the block device file is opened (`open()`), the last reference

```
struct cr_disk_s {

     char             name[CR_MAX_DISK_NAME_LENGTH];
     struct list_head list;
     atomic_t         flags;
     uint32_t         ID;          // the device ID
     int              type;        // 0: data, 1: redundancy
     int              operational; // 0: no,   1: yes
     kdev_t           dev;         // device number
     kdev_t           ctrldev;     // the control device,
};
typedef struct cr_disk_s cr_disk_t;
```

FIGURE 5.6.: The ClusterRAID disk type.

```
struct cr_bh_s {

     atomic_t            remaining;
     atomic_t            succeeded;
     atomic_t            failed;
     int                 master_uptodate;
     kdev_t              data_dev;
     struct buffer_head *master;
     struct buffer_head *copy_bh_list;
     struct buffer_head *data_disk_bh;
#ifdef COPYMODE
     struct buffer_head *check_bh;
#endif
};
typedef struct cr_bh_s cr_bh_t;
```

FIGURE 5.7.: The ClusterRAID type for meta buffer heads.

```
struct cr_bh_container_s {

     int      type;                     // 0 = data, 1 = redundancy
     int      rw;                       // 0 = read, 1 = write
     struct   buffer_head *bh;
     struct   buffer_head *read_bh;
     void   (*b_end_io)( struct buffer_head *bh, int uptodate);
     char    *b_data;                   // pointer to data block
     struct   page *b_page;
     struct   list_head list;
     atomic_t rlocked;
#ifdef COPYMODE
     struct   buffer_head *check_bh;
#endif
};
typedef struct cr_bh_container_s cr_bh_container_t;
```

FIGURE 5.8.: The ClusterRAID type for buffer head containers.

to it is closed (`release()`), or an ioctl (`ioctl()`) is issued to the device. In the ClusterRAID driver the `cr_open()` and `cr_release()` functions only increment and decrement the module usage counter. The `cr_ioctl()` method is mainly used as a control interface, as will be described in section 5.5. The remaining functions of this structure holding block device operations are not implemented, as they are not needed for the specific operation of the ClusterRAID. Upon module loading the `cr_init()` function of the driver registers the block device operations, and upon unloading the function `cr_exit()` performs the unregistering. In addition to the general block device driver registration and deregistration, these functions also perform the creation and deletion of all ClusterRAID kernel threads, the creation and removal of the `/proc` entry, the allocation and deallocation of private caches, and the necessary modification of all ClusterRAID related kernel tables. Examples for these tables are `blksize_size[]` or `hardsect_size[]`, which describe respectively the block size and the sector size used by the devices registered with the kernel.

Functions involved in request handling and callbacks are closely related. While the former usually arrange the transfer of a given block, they choose a callback function that is invoked after the transfer has been serviced by the underlying device. The handling of requests and the role of the callback functions will be examined in detail during the functional analysis in section 5.3.

Since the ClusterRAID driver uses several threads, e.g. to serve write requests (`kcrd`) or for the reconstruction of data (`kcrrecoveryd`), their creation, initialization, and deletion is encapsulated into a group of thread-related functions, which are inspired by the Linux kernel threads tutorial [158]. Along with a brief functional description, the methods used inside the ClusterRAID driver are depicted in table 5.1.

The driver provides a pair of functions which can be used by codec modules to register with the main module, namely `cr_[un]register_redundancy_algorithm()`. While the unregister function only has one parameter, notably an ID, which identifies this specific codec module and its algorithm, the register function must additionally provide an interface structure that stores references to the implemented coding and decoding functionality.

5.2.2. The Codec Module

The codec module `cr_rs` is a kernel implementation of the Vandermonde-based Reed-Solomon codes introduced in section 4.5. In order to keep the main module independent from the coding algorithm, the ClusterRAID design requires codecs to be separated into dedicated modules. Furthermore, this encapsulation will ease the port of the whole driver suite to newer versions of the kernel, since the codec module is only accessed by the main module via a defined interface, and – aside from registering with the core kernel – requires no further interaction with it. Hence, the effort of adapting to new kernel releases is minimized for this part of the ClusterRAID functionality.

A codec module can register itself with the main module by means of the type `cr_redundancy_algorithm_t`, shown in figure 5.9. The main module can have several codec modules registered at the same time. Each ClusterRAID *device* can of course only deploy one codec at a time, and all devices that share redundancy areas must deploy the same codec with the same configuration in order to produce meaningful redundancy information.

In the interface structure, the field `name` is used to identify a specific algorithm in debugging and error messages. It is also used as the codec's identifier by the ClusterRAID's `/proc` interface. The `get_handle()` function pointer is invoked to generate an instance of the codec. This handle

Function Name	Parameters	Description
`cr_create_thread()`	`void(*fn)(cr_kthread_t *),` `cr_kthread_t *cr_thread`	Generate a kernel thread. The function pointer determines the actions of this thread.
`cr_delete_thread()`	`cr_kthread_t *cr_thread`	Delete a kernel thread by sending a KILL signal.
`cr_thread_launcher()`	`void *data`	Interface to the kernel's thread generator function `kernel_thread()`.
`cr_init_thread()`	`cr_kthread_t *cr_thread,` `char *name`	Initialize a kernel thread. This function is invoked once from within the thread function `fn`, usually before entering the main loop.
`cr_exit_thread()`	`cr_kthread_t *cr_thread`	Exit a kernel thread. This function is invoked once from within the thread function `fn`, usually after leaving the main loop.

TABLE 5.1.: Thread-related functions in the `cr_main` module.

contains the actual codec configuration and is used by the individual ClusterRAID devices as a parameter for the encoding and decoding functions. As will be shown later, a ClusterRAID device can have multiple handles to allow for simultaneous processing of data in order to speed up processing on SMP computers. By means of the `release_handle` function pointer, all resources allocated with the handle can be freed again. The `get_info` function pointer provides access to information about the configuration of a codec instance, such as the total number of devices, the number of redundancy devices, and the number of tolerable failures. The two central functions, however, are accessed by the `encode_block` and `decode_block` function pointers. As their names suggest, these function pointers respectively encode a data block, usually during a write, or reconstruct data in case of failures. While the above-mentioned `get_handle` generates a new handle with default parameters, a codec can be dynamically reconfigured during runtime with new parameters using the `reconfigure` function pointer. In figure 5.9 the parameters of the functions are omitted for reasons of clarity. All functions expect a handle identifying the codec instance as their first parameter, except for `get_handle()`, which generates and returns such a handle. The type `cr_rs_t` as currently

```
struct cr_redundancy_algorithm_s {

     char      *name;
     cr_rs_t *(*get_handle)();
     void     (*release_handle)();
     int      (*get_info)();
     int      (*encode_block)();
     int      (*decode_block)();
     int      (*reconfigure)();
};
typedef struct cr_redundancy_algorithm_s cr_redundancy_algorithm_t;
```

FIGURE 5.9.: The ClusterRAID's structure for a redundancy algorithm. The parameters of the individual interface functions are omitted for reasons of clarity.

```
struct matrix_s {

     uint32_t   rows;
     uint32_t   columns;
     uint16_t **entries;
};
```

FIGURE 5.10.: The structure `matrix_s`.

being used is adapted to the Reed-Solomon based codec, but can be easily extended and adapted to be suitable for other codec implementations as well. Its inner structure is set out in the next section. The functions of the codec module can be categorized into three main classes, which will be discussed in the following paragraphs, along with the data structures used.

Internal Data Structures

In addition to the structure used as the interface to the main module, the codec module uses only two data structures. As exposed in section 4.5, matrices are the fundamental structure on which the algorithm is based. The simple structure `matrix_s` shown in figure 5.10 contains fields for the matrix dimension (`rows` and `columns`) and a pointer to an array of the matrix elements (`entries`). The size of the entries of 16 bits reflects the maximum symbol size the module supports. This value is a trade-off between enlarging the symbols to limit the computational overhead on the one hand and a suitable size of the exponential and logarithmic lookup tables for Galois field elements on the other. In addition to 16-bit symbols, the driver also supports 8-bit symbols.

The other data structure used holds all information about specific instances of the codec. The elements the type `cr_rs_t` comprises are listed in figure 5.11. The matrix stored by `idm` is the information dispersal matrix F' used for data encoding, as defined by equation 4.44. The matrix used to reconstruct data in case of failures is referenced by `dm`, i.e. the decode matrix. Data reconstruction can be put down to solving a set of linear equations over a Galois field. The codec module implements two solvers, one based on the LU decomposition of the decode matrix, the other based on the inverse of the decode matrix. Both solvers rely on the in-place decomposition as given by Crout's algorithm [159]. Depending on the solver, the pointers `dmLU` and `dmInv` are used for the

```
struct cr_rs_s {

     struct matrix_s *idm;       // information dispersal matrix
     struct matrix_s *dm;        // decode matrix
     struct matrix_s *dmLU;      // LU decomposition of dm
     struct matrix_s *dmInv;     // inverse of dm
     struct matrix_s *wm;        // work matrix
     int32_t         *index;     // permutation of the rows during
                                 // LU decomposition
     int32_t          failed;    // unique identifier of failed devices
                                 // combination
     uint16_t         symSize;   // the symbol size this handle is
                                 // configured for
     uint16_t         NN;        // number of code symbols per block
                                 // (data+redundancy)
     uint16_t         PP;        // number of redundancy devices
     uint16_t        *log;       // log LUT
     uint16_t        *exp;       // exp LUT
     uint16_t        *vector;    // the b-vector  (Ax=b)
     uint16_t        *vectorLU;  // the result vector for the
                                 // reconstruction using the LU
                                 // decomposition
     uint16_t        *vectorInv; // the result vector for the
                                 // reconstruction using the
                                 // inverse matrix
};
typedef struct cr_rs_s cr_rs_t;
```

FIGURE 5.11.: The type `cr_rs_t`.

LU decomposition and the inverse, respectively. It is important to note that both matrices need to be calculated only once for a specific combination of missing devices. During the LU decomposition it may be necessary to interchange certain rows of the matrix. This row permutation is tracked by the `index` array, as this information is required later for the LU substitution. The `failed` field holds a number unique to a specific combination of faulty devices. If the number changes, for instance due to the return to operation of one of the faulty devices, or due to another device breaking during reconstruction, the decode matrix must be recalculated. The fields `symSize`, `NN`, and `PP` hold information about this instance's symbol size, the total number of devices, and the number of redundancy devices it is configured for, respectively. The references to the lookup tables of the exponential and logarithmic values of the Galois field used for this codec are accessed by `exp` and `log`, respectively. The remaining fields `vector`, `vectorLU`, `vectorInv`, and `wm` are used for temporary storage of intermediate results.

The Galois Field Functionality

The codec module provides a set of functions dedicated to the algebra over Galois fields, on which the algorithm is based. The actual implementation of these functions follows the proposed implementation given in [148, 149]. Table 5.2 lists the relevant methods in combination with a brief description.

Common to all functions is the parameter `handle`, which is a pointer to a `cr_rs_t` structure that holds all information about the codec instance. For instance, for the multiplication or division of

Function Name	Parameters	Description
`gf_mult()`	`cr_rs_t *handle, int a, int b`	Return the product of the two Galois field elements `a` and `b`.
`gf_div()`	`cr_rs_t *handle, int a, int b`	Return the quotient of the two Galois field elements `a` and `b`.
`gf_pow()`	`cr_rs_t *handle, int a, int b`	Return the value of Galois field element `a` raised to the power of Galois field element `b`.
`gf_LU_decomposition()`	`cr_rs_t *handle, struct matrix_s *matrix`	Perform an in situ LU decomposition of `matrix` over the Galois field.
`gf_LU_substitution()`	`cr_rs_t *handle, struct matrix_s *matrix, uint16_t *vector`	Solve the set of linear equations formed by the LU decomposed `matrix` and the right hand side `vector`. The result is again stored in `vector`.
`gf_matrix_inversion()`	`cr_rs_t *handle, struct matrix_s *matrix`	Determine the inverse of the `matrix`. The result is stored in the field `dmInv` of the handle.

TABLE 5.2.: Galois field methods of the codec module.

two Galois field elements the lookup tables for the exponential and logarithmic functions are required. While all functions are needed for the reconstruction of data, only the multiplication function `gf_mult()` has to be used for data encoding.

The Interface Implementation

The implementation of the interface to the main module, as given by figure 5.9, forms the core functionality of the codec module. The initial generation of a codec instance with default parameters is provided by `cr_rs_generate_handle()`. This function calls `cr_rs_init()`, the main function for codec initialization. Invoked with the symbol size, the total number of ClusterRAID devices, and the number of tolerable failures, it allocates all memory required for the fields of a handle. In addition, the lookup tables needed for Galois field multiplication and division as well as the information dispersal matrix are initialized by `cr_rs_setup_tables()` and `cr_rs_idm()`, respectively. The counterpart to the generator function is `cr_rs_release_handle()`, which frees all allocated resources. The configuration parameters of a codec are returned by `cr_rs_provide_info()`. The encoding function in this codec's implementation is `cr_rs_calc_redundancy_diff()`. It encodes the difference of already stored and new data and returns the correction of the already stored redundancy information. The encoding itself is an iteration over all symbols in a given data block and the multiplication of each symbol over the Galois field with a row of the information dispersal matrix. Which row has to be used is determined by the ID of the data node in the ClusterRAID system.

The implementation of the decode function `cr_rs_decode_block()` is more intricate due to its underlying concept of solving a set of linear equations over the Galois field. Basically, the task is broken down to the reconstruction of one redundancy ensemble, i.e. of one set of data and redundancy blocks. Upon the arrival of the first ensemble to be reconstructed, the decode matrix is initialized according to the current combinations of failures by calling `cr_rs_dm()`. As outlined above, the decode matrix may be changed if additional devices fail or if devices become operational again. Depending on the solver algorithm, the solution can be calculated from equation 4.44 either by multiplying the equation with the inverse of the information dispersal matrix F' or by LU substitution. Since the approach using the inverse matrix reduces the task of solving the set of linear equations to Galois multiplications, the additional effort to determine the inverse of the dispersal matrix from its LU decomposition is justifiable. This applies even more, as the matrix inverse will usually be determined relatively infrequently compared to the number of redundancy ensembles to be reconstructed.

The function `cr_rs_reconfigure()` cleans up a given handle and initializes it, again invoking the `cr_rs_init()` function. Table 5.3 on page 118 summarizes the discussed interface functions by showing their call parameters and giving an executive summary of their functionality.

All remaining methods in the codec module are either used for monitoring and debugging or they encapsulate supporting functionality, such as memory management or special matrix operations. Therefore, a detailed description of these functions is omitted.

5.2.3. The XOR Module

The XOR module uses code from the module of the same name which is shipped with the Linux kernel and used by the RAID drivers. It makes use of the Streaming SIMD (Single Instruction Multiple Data) Extensions of a processor to speed up the XOR'ing of data blocks [24, 25]. Not all processors support this feature. Therefore, and since it is an optional optimization, it has been encapsulated to a separated module. Currently, the only function exported by this module is `xor_block()`. This function receives two buffer heads and combines their data by the logical XOR operation. The result is stored to the data buffer of the first buffer head. Since calculating the difference between two data buffers is required multiple times for every write request, employing dedicated entities of the processor with inherent parallelism improves the ClusterRAID performance.

5.3. Functional Analysis

This section will detail the data path for the two request types, read and write, for both normal operation as well as operation in degraded mode. UML sequence diagrams will be used to illustrate the chronology of events and operations.

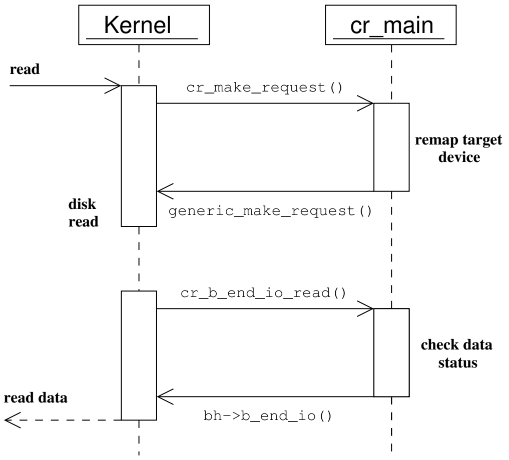

FIGURE 5.12.: UML sequence diagram for successful data read requests.

5.3.1. Read Requests

For reads, the data path inside the ClusterRAID is short. From the buffer head passed to the `make_request_function()` the minor device ID is extracted in order to determine which ClusterRAID device the corresponding request was intended for. The device ID is used to obtain the correct device descriptor from the global array `g_cr_devices[]`. This step is necessary for all request types and also for both data as well as redundancy devices. The function then changes the target device (`b_dev`) from the current ClusterRAID device to the constituent target device, as described by the `data_dev` field of the ClusterRAID device descriptor. The original `b_end_io()` function pointed to by the buffer head's field `b_end_io` is replaced by a reference to the ClusterRAID's callback function for read requests, namely `cr_b_end_io_read()`. After that, the request is reissued by invoking `generic_make_request()`. At this point, the function `cr_make_request()` returns and the ClusterRAID driver releases the remaining CPU time to the core kernel.

Once the constituent data device has serviced the request, the driver is notified via the kernel's call of the implanted callback function. If the `status` parameter passed to this callback function signals a successful completion of the operation, the device descriptor in the buffer head is restored to the ClusterRAID device and the original `b_end_io()` function is called to signal to the kernel that the requested data is now available from the buffer head.

This sequence of events is depicted in the sequence diagram in figure 5.12. The class role objects in this case are the core kernel and the main module of the ClusterRAID driver. The initial message the kernel receives is the read request of an application wanting to access the ClusterRAID device. As described in section 5.1, the kernel invokes the ClusterRAID driver's `make_request_fn()` function of the request queue descriptor. After the request has been reissued by the main module, the driver is finished with the request for the moment. The kernel forwards the request to the data disk and waits for the read to be serviced. Once the hard drive has transferred the requested data to the kernel, the kernel notifies the main module, which checks if the transaction was successful. If this is the case, the kernel is signaling that the driver has serviced the request and that the read data can be handed back to the requester, i.e. the application.

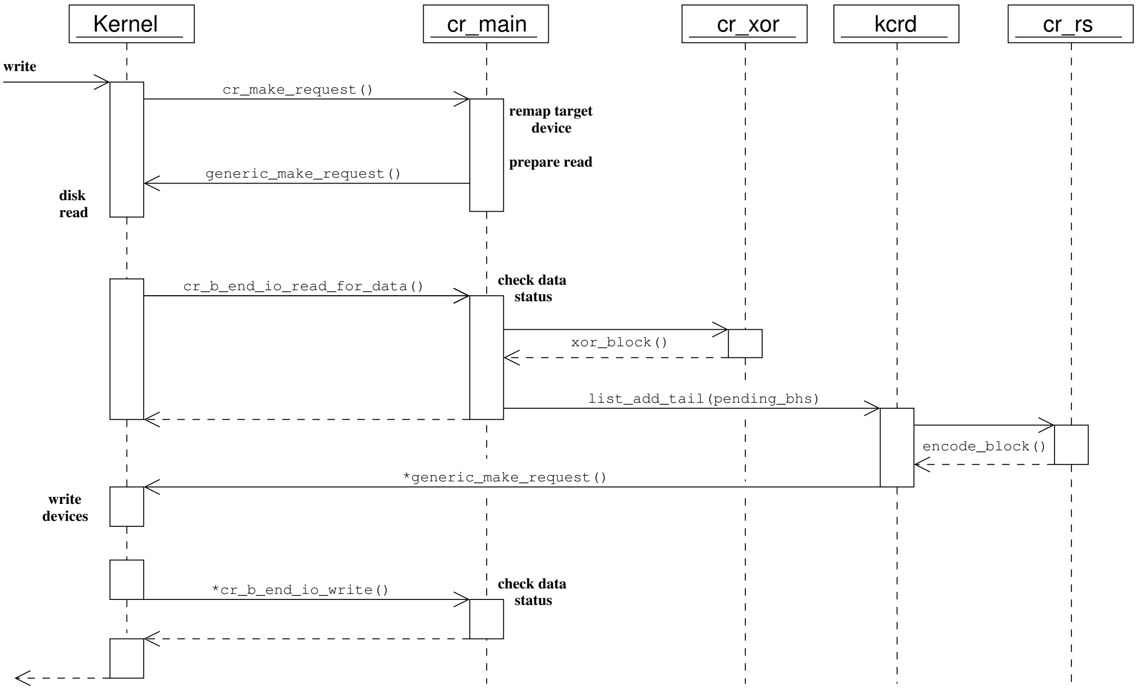
Kernel
cr_main
cr_xor
kcrd
cr_rs
write
cr_make_request()
remap target device
prepare read
generic_make_request()
disk read
cr_b_end_io_read_for_data()
check data status
xor_block()
list_add_tail(pending_bhs)
encode_block()
*generic_make_request()
write devices
*cr_b_end_io_write()
check data status

FIGURE 5.13.: UML sequence diagram for successful data write requests.

If the constituent device stores redundancy information, the procedure is almost identical. The device that the request is forwarded to is the redundancy device in the device descriptor `redundancy_dev` and the callback function is now `cr_b_end_io_read_redundancy()`. As the buffer head itself does not hold any information about whether it is intended for a data or for a redundancy device, the distinction has to be made by the different callback functions to be invoked after request serving. This distinction is necessary for the different behaviours of the two device types in case of failures.

5.3.2. Write Requests

Write requests are more intricate than read requests, as they require the generation of redundancy information and the interaction with remote nodes. This increased intricacy is reflected in the sequence diagram for successful write requests, shown in figure 5.13. When an application writes data to a ClusterRAID device, the driver's `cr_make_request()` function is passed a buffer head along with a write flag. Write requests require many more resources than reads. In order to limit the resources allocated by the driver, the number of pending write requests is checked to see whether they exceed a given threshold. If this is the case, the driver blocks new requests and attempts first to flush the requests that are currently being serviced. As soon as the driver is ready to accept new requests, the corresponding block on the device is locked. Since the modification to the redundancy information is calculated from the difference between the data to be written and the data already stored on the device, this locking step is necessary to avoid race conditions during the handling of write requests. These race conditions will occur if the data of two write requests are compared with the same reference data on the data device, instead of the second write being compared with the data of the first. The result will be an inconsistency between data and redundancy information.

In order to obtain the data of the corresponding block from the underlying device, a request must be generated. For this request, a read buffer head is initialized and targeted at the constituent device. The original request is stored in a buffer head container and attached to the `b_private` field of the read buffer head. The callback field of the read buffer head is set either to the function `b_end_io_read_for_data()`, if the underlying device is for data storage, or to `b_end_io_read_for_redundancy()`, if the underlying device is for redundancy information. After setting the callback field to one of these two functions, the buffer head is issued to the hard drive by invoking `generic_make_request()`. For this, two different callback functions are necessary: for a data write request the original content of the buffer head is sent down to the device, in case of a redundancy device this content is XOR'ed with the currently stored data before it is issued again. In addition, the behaviour in case of failures also varies between the two device types, as will be illustrated later during the discussion of the degraded mode.

Upon the invocation of the callback function, the status of the result of the read access is checked. If the data was successfully read from the device, the content of the block is used to determine the XOR difference with the data of the original write request. This processing can either be performed by the internal `cr_xor_blocks_no_malloc()` function or by the `xor_blocks()` function exported by the XOR module for this purpose. In either case, the result is stored in the buffer head container that already holds the reference to the original write request. At this point, the read buffer head is no longer needed and is thus released. The `list` field of the buffer head container is now used to insert it into the ClusterRAID's list of pending write requests for further processing by the ClusterRAID kernel daemon `kcrd`. After waking up this thread, the callback function returns.

The `kcrd` kernel daemon sleeps on its wait queue until either a timeout occurs or until it is woken up by other parts of the driver. If it is not asleep, the thread checks its `pending_bhs` list for buffer head containers with pending write requests. As long as this list contains elements, the daemon invokes the `cr_serve_requests()` function. This function arranges the actual data transfers with the underlying devices. Since the buffer head container already includes the XOR'ed data, the data transfer with an underlying redundancy device is simple: `generic_make_request()` is called with the original buffer head, which now contains the XOR'ed data and also already knows the correct callback function `cr_b_end_io_write_parity()`. The buffer head container is deleted from the list of pending requests and can be released. These steps are performed by the helper function `cr_write_redundancy_only()`. The callback function invoked after the requests have returned from the underlying device simply forwards the request status back to the original requester after it has unlocked the block again.

If the request is for a data device, the `cr_serve_requests()` function instantiates a meta buffer head, in which the `master` buffer head reference is set to the original write buffer head. In addition, buffer heads are allocated for the data device itself as well as for all redundancy devices. The buffer head for the data device is filled using the references in the original buffer head. For each of the redundancy buffer heads a new buffer must be allocated. The references to each of these buffers, a reference to the XOR'ed data, and the device ID are passed to the coding function `encode_block()` of the codec module registered with this ClusterRAID device. In the prototype implementation this is the codec module `cr_rs`. The reference to the meta buffer head is stored within each buffer head. This is necessary in order to be able to determine the original request that these requests refer to. The meta buffer head also keeps track of the number of requests that must be returned using its field `remaining`. The initial value of this field may differ between requests depending on the number of operational devices. It is hence set separately for every meta buffer head. As also indicated in figure 5.13, the requests are then issued to the respective devices by repeated calls of `generic_make_request()`. If there are no more pending write requests in the list `pending_bhs`, the daemon wakes up potentially sleeping processes that are waiting for resources to become free again.

In the callback function for write requests, notably `cr_b_end_io_write()`, the returned write requests are checked for their completion status. Upon the return of all dispatched requests, the callback function for the meta buffer (`cr_end_bh_io()`) is invoked. This function releases the block lock, frees the allocated resources, and invokes the callback function of the original request to signal to the kernel that the request has been completed.

5.3.3. Degraded Mode

The two previous subsections examined the courses of action during the handling of read and write requests by the ClusterRAID driver. In these subsections it was assumed that all devices were operational and no errors occurred during data transfer. One of the main design goals of the ClusterRAID, however, is to have provisions for failures. As the driver is an intermediary layer embedded in the core kernel, it does not deal with failures directly. Instead, the aforementioned callback functions provide the interface that allows the underlying layers to notify the ClusterRAID of any hardware failures. As both real and virtual devices, i.e. local disks and network block devices respectively, are handled through a block device interface, the failure communication always takes place via the callback functions. If a device encounters a failure, the `status` parameter passed to the corresponding callback function is used to inform the ClusterRAID driver about the failure. Until further notification, the driver marks the erroneous device as faulty and, if possible, carries out its operations without using the device. The state of a ClusterRAID device with one or more faulty constituent devices is described as being in *degraded mode*.

Given the three device types, i.e. local data, remote data, and redundancy devices, there are several different failure scenarios. Each of these scenarios requires appropriate handling by the ClusterRAID driver. The most important of these is that of the reconstruction of data after failure of the local data device. This scenario is required to provide data to applications when the local data device is broken. However, this scenario also applies to the case where a faulty data node must be replaced by a spare node. This spare node takes over the faulty node's identity within the ClusterRAID system. The data of the faulty node must then be reconstructed and stored on the new node.

Figure 5.14 shows a sequence diagram for the behaviour of the ClusterRAID if a read request cannot be served from the underlying local data device. The entry point is the callback function for reads, notably `cr_b_end_io_read()`. An error is reported to this function during the processing of a request by the data device. After the function has identified the affected ClusterRAID device, it marks the device as faulty, preventing subsequent requests from being sent to the underlying faulty device. After marking the device, the faulty request is simply reissued to the ClusterRAID device. This avoids any further processing of this request from within the callback function. When the driver receives requests for a faulty data device, it enqueues them to a list with requests that require a data reconstruction. The list `recovery_bhs_d0` is checked regularly by the `cr_recovery0` recovery daemon. The main tasks of this thread are to acquire remote locks and to gather all the blocks re-

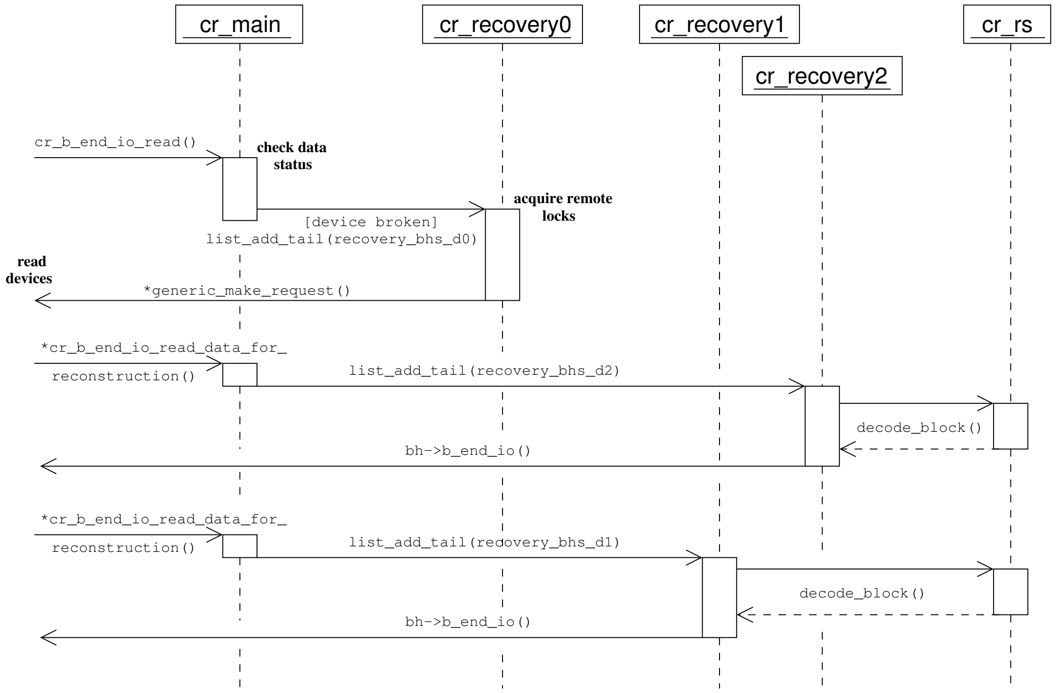
cr_main
cr_recovery0
cr_recovery1
cr_recovery2
cr_rs
cr_b_end_io_read()
check data status
[device broken]
list_add_tail(recovery_bhs_d0)
acquire remote locks
read devices
*generic_make_request()
*cr_b_end_io_read_data_for_reconstruction()
list_add_tail(recovery_bhs_d2)
decode_block()
bh->b_end_io()
*cr_b_end_io_read_data_for_reconstruction()
list_add_tail(recovery_bhs_d1)
decode_block()
bh->b_end_io()

FIGURE 5.14.: UML sequence diagram for a read request in degraded mode.

quired to reconstruct the requested but currently unavailable data. Similarly to all other threads used in the ClusterRAID driver, the daemon `cr_recovery0` usually sleeps until either a sleep timeout occurs or it is woken up by other parts of the driver. If the thread finds entries in its list, it invokes its worker function `cr_reconstruct_block()`, which acquires the remote locks for the corresponding blocks, a procedure which is covered in detail in section 5.4.1. Once it obtained the necessary locks, the function initiates the gathering of the redundancy and remote data blocks required for decoding. The function first initializes an instance of a buffer head container structure into which all the blocks gathered from the remote devices will be inserted. This container also references the array `pos[]` which contains the information about all currently inoperative devices. This information is required during the decoding, and it can change from block to block, e.g. due to additional device failures or the return of repaired devices. Initially, however, the only entry in this array is the local data device that triggered the failure procedure. All devices that must be read from are allocated a buffer head. After this, requests are issued in a similar fashion to a read prior to a local data write, or to reads prior to the issuing of the encoded data in the case of writes on redundancy devices. If devices are marked faulty, no requests are issued to them. In addition, the respective unused buffer heads are released again. The buffer heads sent out to the constituent devices all reference the same callback function, namely `cr_b_end_io_read_data_for_reconstruction()`. If a request is served by one of the requested devices and this callback function is invoked, the counter for the number of remaining buffer heads is decreased and the buffer head is added to the list `bh_list` of the buffer head container. As soon as all requests have returned, the container structure is added to the list `cr_recovery_bhs`, which contains buffer heads prepared for reconstruction. If a request has failed, it is tagged as containing no valid data. In order to speed up the decoding step on SMP systems, the ClusterRAID driver provides a compile time option to use up to four decoding threads instead of one. Where multiple decoding threads are used, the callback function distributes the incoming requests according to a scatter function among them. As the performance of the decoding depends upon how the requests are scheduled to the threads, the actual distribution can be influenced by a simple scatter function, namely `cr_scatter_round_robin_n()`. The gain in performance if the requests are not distributed on a one-by-one basis is a consequence of the fact that – as for all other ClusterRAID daemons – the recovery daemons fall asleep if their request list is empty. Scheduling larger sequences to the threads reduces their sleep-work turn-around cycles.

The recovery threads check their lists with prepared requests at regular intervals. However, before the daemon starts the actual data reconstruction, two cases must be intercepted. With the initial gathering of the remote blocks, some other devices may have been known to be broken *before* and no requests would have been issued to them. As the decoding algorithm expects an array of data buffers, each representing one of the devices in the ClusterRAID system, additional buffers have to be allocated for this case of broken devices for which no buffer head has been issued. Secondly, a device could have failed *after* the requests have been issued and the corresponding buffer heads carry that failure information with them. In this case the array `pos[]` must be updated accordingly.

If the number of valid buffer heads with remote data and redundancy information is sufficient, the decoding can be performed. Otherwise an I/O error must be issued.

As figure 5.5 on page 94 indicated, each ClusterRAID maintains an array of private codec handles. Every recovery thread requires one of these handles for its exclusive use, as it contains buffer space for temporary results during the reconstruction. Additionally, this exclusive handle is needed because the specific constellation of failed devices may vary for different requests. Therefore it is necessary that the daemons maintain their private copy of a handle. This handle can then be adapted on a request-by-request basis to avoid incorrect reconstruction results. As outlined above, the decoding of the data is carried out by the codec module. This module receives an array of buffers, each of which contains one data block. The reconstruction is then performed in an orthogonal fashion according to the individual redundancy ensembles. The data layout as passed to the decoder is depicted in figure 5.15.

As the buffer that represents the faulty data device is included in the array of buffers passed to the codec module, the function `b_end_io()` of the original buffer is invoked after the codec module has reconstructed the data. After all resources allocated to reconstruct this block have been released, the recovery daemon is ready to serve the next request.

The ClusterRAID architecture also allows *writing* to a faulty device. For this however, the data of the preceeding read must be reconstructed, as it is needed to determine the difference to the new data. In order to do so, the prototype implementation uses the same data path as for degraded reads. The requester of the read transactions is the ClusterRAID driver itself. Thus, after reconstruction, the data is not sent back to an application, but is further processed within the main module instead. The buffer head container has a special field `rw` that tags a request as either a read or a write access, see also section 5.2.1 and figure 5.8. If the data has been reconstructed and the original request was a write, a `cr_bh_container` is allocated and the callback for preceeding reads (`cr_b_end_io_read_for_data()`) is invoked, just as if the initial read had succeeded. The recovery daemon hence simply connects the code for reconstruction with the code segments for writing out requests. If devices fail during writes, this will be noticed in the corresponding callback (`cr_b_end_io_-`

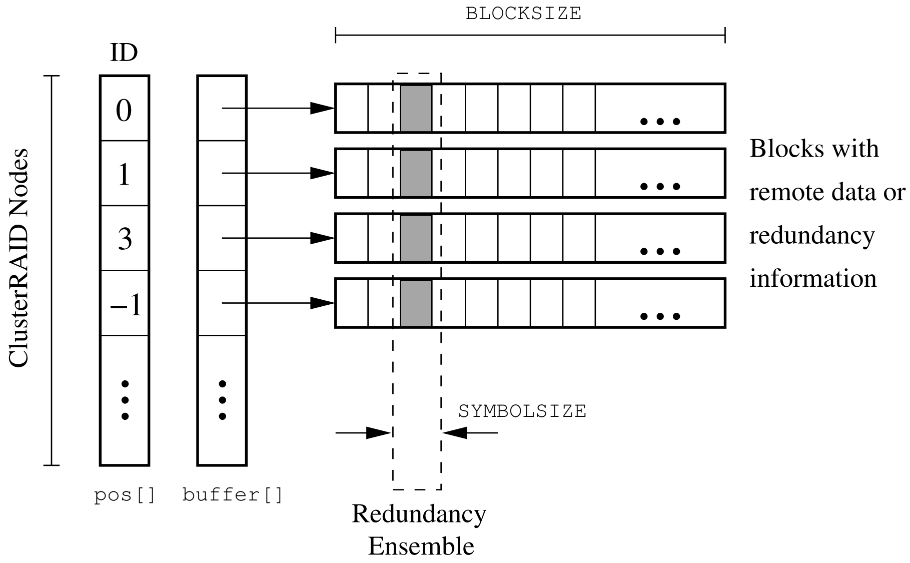

FIGURE 5.15.: Schematic view of the data layout as prepared for the reconstruction. Data buffers that contain no valid data are noted in the array `pos[]`. The actual reconstruction is performed over the redundancy ensembles, orthogonal to the data layout.

`write()`). A device that could not respond to the request is marked as faulty. The number of successful writes to the redundancy devices determines if the write succeeded or if an I/O error is escalated to the kernel, in accordance with the behaviour diagram depicted in figure 3.4. Although the driver allows for writing to a broken device, this may not be desirable for two reasons. Firstly, the resources required by the reconstruction may be needed for other purposes. Secondly, because the current status of the data is to remain unchanged until recovery. Therefore, the ClusterRAID prototype provides a switch to suppress writes to non-operational devices.

5.4. Additional Concepts

This section will cover some of the additional features and concepts of the prototype implementation that have been omitted in the preceeding functional analysis for reasons of clarity.

5.4.1. Locking

Locks are needed in several situations in the ClusterRAID driver – aside from the simple spin locks used to protect shared resources from simultaneous access. As indicated in section 5.3.2, the ClusterRAID driver blocks write buffer heads if there is already a pending write request for that block. This is necessary in order to guarantee consistency between data and redundancy. Therefore, the function `cr_lock_one_block()` is invoked whenever a write request enters the driver. Locks for all blocks of the constituent data device are maintained in a hash table. The buffer head's block number is used as the hash key. If no entry is found in the table, an entry is inserted and the lock for this block is acquired. Otherwise, the request is blocked and the process is suspended. At regular intervals the table is checked to determine whether the lock has been released. This unlocking is carried out by

```
struct cr_block_lock {

     unsigned long       key;
     struct cr_hash_ptrs ptrs;
     atomic_t            used;
     atomic_t            timestamp;
};
```

FIGURE 5.16.: The structure `cr_block_lock`.

calling `cr_unlock_one_block()`. The release happens in the callback functions after the block has been written to the underlying device.

In addition to local locks, the ClusterRAID must allow the acquisition of locks on remote nodes. This is necessary for data reconstruction. The recovery of data can only succeed if the data and the redundancy information read from remote nodes are in a consistent state. If, for instance, a redundancy ensemble is read *after* a data block has been written, but *before* the corresponding redundancy information has reached the redundancy device, the lost data will not be reconstructed correctly. For every block to be reconstructed all pending writes on the remote nodes must be flushed and no other write requests *for this specific block* can be accepted in the meantime on the data nodes. Basically, for remote locking the same mechanism as for local locks is used. For an application accessing a data node there is no difference between the case that the block is locked because of a pending write or because of a remote reconstruction. However, as remote locks are only required for reconstruction, the ClusterRAID does not lock only one block, but sets of consecutive blocks. The number of blocks is defined by the global `LOCKWINDOW` variable. This window locking approach improves the speed of reconstruction and reduces the overhead for remote locking. However, more blocks than necessary are locked by this approach. This is a tradeoff between the locking overhead and the size of blocked areas during reconstruction.

In the ClusterRAID prototype, remote lock acquisition is implemented by lock devices. For this purpose, every node exports a lock device. Reads to specific offsets within a block are interpreted by the remote driver as locking and unlocking commands. The order in which remote locks must be acquired is fixed in a ClusterRAID system. This avoids that multiple reconstructing processes block each other. Once a node has acquired the necessary locks on all involved nodes, it can commence the reconstruction of this block window. If there are multiple recovery daemons that decode data, they can use the same lock simultaneously. Therefore, besides the field `key` for the hash key and the field `ptrs` for the insertion into the hash, the structure `cr_block_lock`, depicted in figure 5.16, comprises a field `used` for the simultaneous lock usage. It denotes the number of concurrent users and protects the release of the lock by one daemon whilst another daemon still needs it. Once `used` drops to zero it is possible to release the lock. However, if multiple daemons are used for decoding, it may happen that one daemon releases a window just before another daemon tries to acquire it. Since the acquiring of locks involves remote read transactions, this acquire-release cycles should be avoided. Therefore, the daemons only mark the remote locks for release, instead of freeing them immediately. The actual lock release is controlled by the `cr_recovery0` daemon running locally on every node. This daemon checks the field `timestamp` of the lock structure and fully releases the lock after a certain amount of time has passed. Tests showed that this feature has reduced unwanted acquire-release cycles significantly.

5.4.2. Copy Mode

The ClusterRAID driver features a compile-time option, named the `COPYMODE`, which forces the internal replication of every buffer head that is issued to the driver for writing. Although this mode increases the computational overhead for write requests, it can become necessary if the ClusterRAID is used together with file systems. Usually, if a block device is accessed, the data is cached in the buffer cache in order to speed up later accesses. Dirty buffers in this cache are flushed to the underlying devices at regular intervals. However, although the dirty buffer is processed by a device driver, it can still be referenced – and thus its content altered – by other parts of the kernel, such as file systems. Therefore, a buffer head's data may be changed after it has been passed to the driver. Hence, in the moment the data block is written to the actual hardware, it will not necessarily contain the same content as when it was issued from the buffer cache. For standard block devices, e.g. hard drives, this does not matter, because the buffer head is always marked dirty after it has been changed. Thus, the changes will either be written to the drive by this transaction or by the next. For the ClusterRAID, however, the content must not be changed after the driver starts processing the buffer, since the difference between newly written and already stored data is used in the redundancy calculation. If the data block is changed after the difference has been determined and the changed data is written to disk, the old difference is used to determine the correction of the redundancy information. Data and redundancy information will be inconsistent.

In order to protect from such data corruption, the ClusterRAID driver copies the data content when it receives the buffer heads in the function `cr_make_request()`. This copied data is then used throughout the entire processing. If the original data is changed during the request handling by the driver, the buffer head will be marked dirty and reissued later as a new request. The ClusterRAID device itself, however, blocks secondary write accesses to this block during the processing, as described in section 5.3.2. Read requests are not affected by this write blocking and can be serviced by the driver.

5.4.3. Kernel Caches

A write request must pass the kernel disk caches, notably the buffer cache and the page cache, on its data path from the application to the constituent data device and the remote redundancy devices. The page cache contains whole pages, each of which corresponds to several logically contiguous blocks of a file or a block device. In analogy, the buffer cache contains buffers, each of which corresponds to a single block. In kernels of the Linux 2.4 series these two disk caches are intertwined in that the amount of memory cached in the buffer cache is always a subset of the amount of memory stored in the page cache [114].

A data request is always stored in the local buffer cache before it is issued to the ClusterRAID. Once flushed to the driver by the memory management system, the redundancy information is calculated and transmitted by the network block device system. On the redundancy side, the data is cached again before actually being written to the redundancy device. The redundancy storage area is usually shared by multiple data nodes, and therefore multiple requests from different data nodes for the same block may arrive closely together in time at a redundancy node. If these requests are cached and one of them overwrites another inside the cache, data and redundancy information are left in an inconsistent state. Therefore, it is mandatory that common caching is avoided on the redundancy side. In the ClusterRAID prototype implementation the redundancy devices are exported to the data nodes by separate logical devices which all target at the same underlying physical device. For the kernel, however, these are now independent devices and accesses to the redundancy device from different data nodes are cached separately. This approach avoids requests from different nodes being overwritten in the buffer cache. However, two consecutive requests from the same data node may also overwrite each other in the buffer cache on the redundancy side. This would have a similar negative effect as overwriting requests from other data nodes. It is therefore in addition necessary that the NBD server writes the redundancy data synchronously to its device.

5.4.4. Private Caches

One approach to reduce the interaction of the ClusterRAID driver with the core kernel is to minimize the allocation and deallocation of memory by deploying private caches. The Linux kernel supports this technique by means of the lookaside cache structure, namely `kmem_cache_t`. Lookaside caches are usually instantiated for frequently used structures in the kernel. Hence, the kernel itself also maintains several of these caches, e.g. for buffer heads, notably the buffer head cache pool `bh_cachep`. This kernel cache is used by the ClusterRAID whenever it must allocate `buffer_head` structures. The primary advantage of using such caches is that the kernel automatically grows and shrinks the cache as appropriate. If, however, drivers set up their own caches using this kernel interface, the cache control remains with the core kernel. Therefore, some drivers, for instance the RAID-5 driver, prefer to manage their own caches. All control is then on the driver's side. The data structures most frequently allocated and freed by the ClusterRAID are container structures, buffer heads, and pages. As the latter represent the largest pieces of memory, and since the deployment of the cache was found to greatly improve the driver's stability, the driver currently only maintains a private cache for whole pages. The number of requests processed by the driver at any point in time is limited. Hence, the maximum number of pages can be derived from the actual configuration. Currently, this amount of pages is aggressively cached on driver initialization and not freed before driver unload. The interface for page allocation inside the driver is provided by `cr_page_alloc()` and `cr_page_free()`.

5.4.5. Check Blocks

All write requests issued to a redundancy device trigger an update procedure realized by a read-modify-write cycle: the data already stored is read, XOR'ed with the correction of the redundancy information, and written back. If a request is split into two pieces, and hence two separate *partial* writes are issued to the same block, the result is not necessarily the XOR of the new and the already stored data. This is due to the actions taken by the kernel whenever only a fraction of a block is to be written: it must read the corresponding block as a whole and then change the requested parts. A sample scenario is depicted in figure 5.17. In the first part of the write transaction, only the left fraction of the block is available, the second fraction is written in the following write. The result of the transactions is always shown in the last line. As these writes are issued by the kernel to the ClusterRAID device, the driver can only check if there are subsequent writes to the same block.

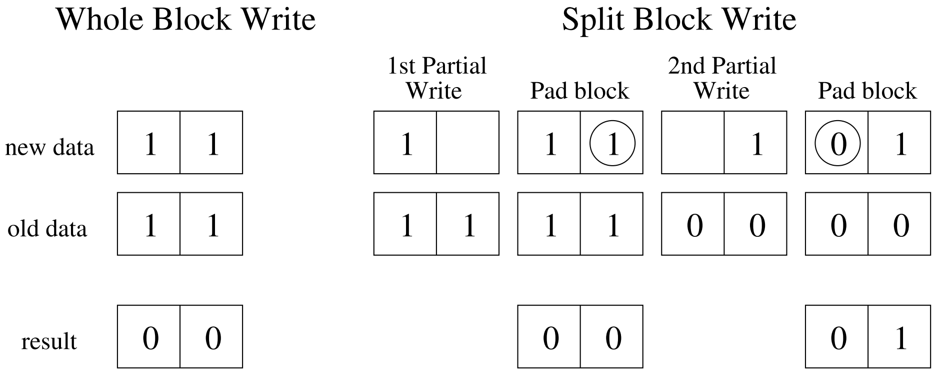

FIGURE 5.17.: Illustration of a split block write: blocks are always written as a whole. The left half of the figure illustrates a typical write of a whole block, the right half a split write. Only a part of the new data is available and the kernel reads the rest of the block from the disk. This padded data is encircled in the example.

Therefore, the driver provides the `CHECKBLOCKS` option for the detection of possibly split requests. Write requests to redundancy devices are initiated only from remote data devices. Therefore, the redundancy device should always receive whole blocks for writing. However, the request is sent over the network and may be split in transit. The network block device server, the application that eventually issues writes to the redundancy device, can always wait for a whole request before writing. With version 2.4.32 the ENBD server has been adapted accordingly [157]. Another approach is to ensure that no requests are split, making sure that the (E)NBD server always receives a whole block. ENBD does this if its `merge_requests` is switched off, NBD does this by default.

On the data device side, partial writes do not cause problems, as the data is written unchanged to the constituent device and the difference used for encoding is simply zero for the rest of the block.

5.5. Monitoring, Debugging, Control, and Configuration

Since the ClusterRAID is a complex system, basic support for monitoring, control, and configuration is provided in the prototype implementation. This support makes use of the `/proc` interface for monitoring, a user library to debug and set up the driver, and an XML-based [160] configuration, start, and control mechanism.

5.5.1. Monitoring

As is typical for kernel drivers, the ClusterRAID provides debugging and status information via an entry in the virtual `/proc` file system, namely `/proc/crstat`. Figure 5.18 shows a snapshot of that information for an active ClusterRAID data device. The `query` counter in line 1 is incremented every time `/proc/crstat` is opened. Line 3 lists the redundancy algorithms registered with the ClusterRAID driver. In this example only the Reed-Solomon codec is registered. The lines 5 through 32 set out information about the ClusterRAID device `cr0` identified by the ClusterRAID ID 3 (line 5). The following subsection (lines 6 through 15) lists all devices of the ClusterRAID system and

```
 1:     ClusterRAID (query 34415)
 2:     ---------------------------
 3:     [Reed-Solomon]
 4:
 5:     cr0 [ID 3]:
 6:                 ID            name          type     status      locality
 7:                 ---------------------------------------------------------
 8:                  3       /dev/hda7          data         U          local
 9:                  0        /dev/nd0    redundancy         U         remote
10:                  1        /dev/nd2    redundancy         U         remote
11:                  0        /dev/nd4     rem. data         U         remote
12:                  1        /dev/nd6     rem. data         U         remote
13:                  2        /dev/nd8     rem. data         U         remote
14:                  4       /dev/nd12     rem. data         U         remote
15:                  5       /dev/nd14     rem. data         U         remote
16:
17:                  algorithm: Reed-Solomon (8,2,2)
18:                  pdisks   : 2
19:                  rdisks   : 5
20:                  faulty   : 0
21:                  req limit: 4096
22:                  BLOCKSIZE: 1024
23:                  locked   : 4098
24:                  lckd win : 0
25:                  pages    : 8226
26:
27:
28:                  READs    37081973,                       done   37081973
29:                  pending         0
30:
31:                  WRITEs   36053042, re-issued   36049118, done   36048944
32:                  pending      4098
```

FIGURE 5.18.: The ClusterRAID driver output under `/proc/crstat` for an active data node.

provides a brief overview of their characteristics: the ID, the name by which the device is accessible from the local host, the device type (data, redundancy, or remote data), the device status, and its location. The lines 17 through 25 describe the actual configuration of the local device: the algorithm used, the number of redundancy devices, the number of remote data devices, the maximum number of pending write requests for the device, the block size used, the number of locked blocks, the number of locked reconstruction windows, and the number of pages currently used in the private page cache. The remaining lines 28 through 32 show the number of read and write requests for this device. The first figure is the number of requests the device has received, the last is the number of served request. Since write requests are always preceeded by a read and are then re-issued, there is an additional number to account for that. In addition, the number of requests that are currently served by the driver is given in the lines starting with `pending`.

If the node only stores redundancy data, the output looks different for reasons of clarity. The sample snapshot in figure 5.19 shows sections for the device nodes `cr0` through `cr5`, which are exported by means of an NBD to each of the data nodes. The ID is the unique redundancy ID of this redundancy node, `1` in this example. This is the same ID used by the data nodes to identify the redundancy node, as shown in line 10 in figure 5.18. Furthermore, the individual sections state the underlying device (`/dev/hda7`), the type, the status, and the locality as in the case of a data node. The number of read and write requests for this particular device are listed, namely the number of received and served requests separated by a forward slash. Finally, there are two more lines indicating the number of

```
ClusterRAID (query 55976)
---------------------------

               ID            name          type       status    locality
               ----------------------------------------------------------
cr0 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    36739410/36739409
     locked:    220
     caches:    yes

cr1 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    39704886/39704886
     locked:    220
     caches:    yes

cr2 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    31581601/31581474
     locked:    220
     caches:    yes

cr3 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    35965650/35965650
     locked:    220
     caches:    yes

cr4 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    34657334/34657242
     locked:    220
     caches:    yes

cr5 [ID 1] :    1       /dev/hda7    redundancy           U        local
     READs :    0/0
     WRITEs:    34469515/34469515
     locked:    220
     caches:    yes
```

FIGURE 5.19.: The ClusterRAID driver output under `/proc/crstat` for a redundancy node.

blocks currently blocked for write access and whether the driver uses private caches.

5.5.2. Driver Control and Configuration

The ClusterRAID prototype driver can be configured via the `mkcr` control program. Communication between this program and the driver is implemented using the `ioctl()` system call. Symbolic links determine which command should be executed. Table 5.4 on page 119 lists the names of the symbolic links and the corresponding actions taken by the driver.

The basic functionality provided is the addition of underlying devices to a ClusterRAID system. For data devices, redundancy devices, and remote data devices the links `d-add`, `p-add`, and `r-add` are used, respectively. Spare devices need not to be inserted into the driver, since the moment they are added to the system, their type changes to data or redundancy. Using `rem`, devices can be removed from the ClusterRAID driver. This may become necessary in the case of a faulty device to be replaced by a spare one. Since the ClusterRAID prototype is not restricted to one specific redundancy algorithm, and the implementation of the main driver provides a registration interface for codec modules, the algorithm to be used can be selected using `setalgorithm`. If no algorithm is selected or registered, the device will simply mirror all write requests to the underlying devices. In addition, a registered algorithm can be switched off, using the `usealgorithm` link. `Setsync` sets the device into synchronous mode, i.e. at all times the driver has at most one single pending request. The device ID of the driver is set by `setdeviceID`. The `reconfigure` link initializes the codec driver selected for the device (if any) and initializes it. In case of the `cr_rs` Reed-Solomon codec, discussed in section 5.2.2, the `reconfigure` link provides the symbol width, the number of data devices, and the number of redundancy devices in the system. To ensure that the devices are in a clean state following their startup, the devices can be initialized with zeroes using the `initcr` link. Although not implemented in the prototype, a ClusterRAID system with valid data on the devices could also be cleansed by using the routines for data encoding. The remaining commands are intended for debugging and test purposes, such as triggering specific failure conditions by `genreadfaulty`, `genwritefaulty`, `setreadfaulty`, or `setwritefaulty`, or the lock handling by `lockblocks` and `unlockblocks`.

5.5.3. ClusterRAID Configuration and Startup

The specific configuration of a ClusterRAID setup is specified in an XML file. The top level node of a well-formed ClusterRAID XML configuration document contains a `<cr2>` tag. This node has an attribute `config` specifying the total number of nodes in the system as well as the number of redundancy nodes. The node `<cr2>` is divided into two parts, a global section with path and port information and a node specific one, which details the configuration of a single ClusterRAID node. A typical ClusterRAID configuration file is given in figure 5.20.

The `hostname` is used to identify the configuration for a specific node. Since a node can have several network interconnects, the hostname to be used to connect to this node is given by `connectname`. The `crID` is the unique data or redundancy node ID of a node in the ClusterRAID. The `type` determines if the node stores redundancy information, user data, or is just a spare node. The `algorithm` node selects the redundancy algorithm to be used with this device. For optimization purposes the ClusterRAID driver can use the SSE instructions of the processor (if supported). This option can be selected using the `cr_xor` node. The remaining `mount` specifies the mount point that the device should be attached to. The configuration file is read in by a Ruby-based [161] parser script `cr2launch.rb`, which is responsible for module loading, driver configuration, the launch of network block device processes, and the mounting of the device. Table 5.5 on page 120 lists the available commands and the corresponding actions of this control script.

Although the configuration and startup control is sufficient for the ClusterRAID prototype, it may be desirable to consider more sophisticated process launch and control mechanisms, such as [162], in future implementations.

```
<?xml version="1.0"?>

<cr2 config="3,1">

  <crpath>/home/wiebalck/cr/</crpath>
  <nbdpath>/home/wiebalck/nbd-2.6-awi/</nbdpath>
  <portstart>5000</portstart>
  <portstride>100</portstride>
  <ctrlport>9999</ctrlport>

  <node>
    <hostname>e007</hostname>
    <connectname>eg007</connectname>
    <crID>0</crID>
    <type>redundancy</type>
    <algorithm>0</algorithm>
    <device>/dev/hda7</device>
    <cr_xor>yes</cr_xor>
  </node>

  <node>
    <hostname>e001</hostname>
    <connectname>eg001</connectname>
    <crID>1</crID>
    <type>data</type>
    <algorithm>0</algorithm>
    <device>/dev/hda7</device>
    <cr_xor>yes</cr_xor>
  </node>

  <node>
    <hostname>e000</hostname>
    <connectname>eg000</connectname>
    <crID>0</crID>
    <type>data</type>
    <algorithm>0</algorithm>
    <device>/dev/hda7</device>
    <cr_xor>yes</cr_xor>
  </node>

</cr2>
```

FIGURE 5.20.: Sample ClusterRAID configuration for a (3,1) setup.

Function Name	Parameters	Description
`cr_rs_generate_handle()`	`void`	Return an initialized handle of a codec with default parameters.
`cr_rs_release_handle()`	`cr_rs_t *handle`	Free all resources of a codec handle.

`cr_rs_provide_info()`	`cr_rs_t *handle,`
	`uint16_t *nn,`
	`uint16_t *pp,`
	`uint16_t *tt`
		Return the information about a codec instance identified by a handle.

`cr_rs_calc_redundancy_diff()`	`cr_rs_t *handle,`
	`uint32_t ddevice,`
	`uint32_t pdevice,`
	`uint16_t *source,`
	`uint16_t *target`
		Calculate the correction of the redundancy information from the data difference stored in `source` for data device `ddevice` and redundancy device `pdevice`, then store the result in `target`.

`cr_rs_decode_block()`	`cr_rs_t* handle,`
	`uint16_t **data,`
	`uint16_t *pos,`
	`int32_t no,`
	`uint16_t id`
		Reconstruct the data block of the ClusterRAID node with ID `id`. The available remote data and the redundancy information can be accessed via the two-dimensional `data` array. The number and position of invalid data is given by `no` and `pos`, respectively.

`cr_rs_reconfigure()`	`uint32_t symSize,`
	`uint32_t NN,`
	`uint32_t PP`
		Reconfigure the `handle` of a codec to work for `NN` nodes, which can tolerate `PP` failures and uses `symSize` bits as the symbol width.

TABLE 5.3.: Interface methods of the codec module.

Command	Action
d-add	Add a data device.
p-add	Add a redundancy device.
r-add	Add a remote data device.
rem	Remove a device.
setalgorithm	Select a registered redundancy algorithm.
usealgorithm	Toggle the flag that determines if the algorithm is used.
setsync	Set the device into synchronous mode.
setdeviceID	Set the ClusterRAID device ID.
reconfigure	(Re-)Configure the algorithm.
setreqlimit	Set the request limit.
setredlvl	Set the redundancy level.
initcr	Initialize the attached devices with zeroes.
setfaulty	Mark a device faulty.
setoperational	Mark a device operational.
genreaderror	Generate an error *during* the next read.
genwriteerror	Generate an error *during* the next write.
setreadfaulty	Generate an error *before* the next read.
setwritefaulty	Generate an error *before* the next write.
lockblocks	Lock specific blocks.
unlockblocks	Unlock specific blocks.
dumplocks	Dump the number of the locked blocks.
syncinv	Sync and invalidate the buffer of this device in the buffer cache.
resetcounter	Reset the counter under `/proc/crstat`.

TABLE 5.4.: Commands and actions of the ClusterRAID control library (`mkcr`).

Command	Action
cr2launch	Load the modules (`cr.o, cr_rs.o, cr_xor`). (Re-)Configure the algorithm. Add the local device. Start the NBD servers.
cr2init	Initialize all added devices with zeroes.
cr2connect	Start the NBD clients and connect to the correct servers. Add the remote data devices and the redundancy devices.
cr2replace	Replace one of the nodes with a spare node. Start the resynchronization of the new node.
cr2kill	Stop the ClusterRAID on the nodes. Unmount the device. Stop all NBD processes. Unload the modules.
cr2info	Parse the XML configuration file. Print the configuration in a clearly arranged way.

TABLE 5.5.: Commands and actions of the ClusterRAID launcher script (`cr2launch.rb`).

6. Prototyping and Benchmarks

In truth, there are atoms and a void.
Democritus

This chapter details the benchmarks as performed with the ClusterRAID prototype implementation. After a brief description of the test setup and benchmarks of the underlying components, i.e. hard disk, network, and network block device, the results of both performance measurements and functional tests will be presented. Among the latter the results of the cooperation of the ClusterRAID with a distributed file system, notably the MOSIX File System, will be presented. The chapter is closed by a discussion of implemented optimizations and possible enhancements to improve the ClusterRAID's performance by further software adaptations or additional hardware support.

6.1. The Test Environment

All performance, functionality, and stability tests presented in this chapter were performed on subsets of the 32 node Cyclops Plant cluster that is installed at the Kirchhoff Institute of Physics [163]. This cluster comprises dual Pentium III 800 MHz PCs with a SuSE Linux [164, 165] operating system version 9.0. The nodes are running a standard kernel version 2.4.23 from [41] with the Precise Accounting Patch [166, 167] applied. Their Tyan [168] Thunder HEsl motherboards are equipped with the Serverworks [169] HEsl chipset. Each node has 512 MB of RAM and a 40 GB IBM DTLA-307045 IDE drive. For the network interconnect, the onboard Intel EtherPro 100 Fast Ethernet and 3com [170] 3C996B-T Gigabit Ethernet PCI cards were used. The switches used throughout the testing were a Trendnet [171] TEGS224M and a Netgear [172] GS508T for Fast and Gigabit Ethernet, respectively.

6.1.1. The Hard Disks

Rather than the CPUs or the network interconnect, the hard drives are the fundamental hardware component of the ClusterRAID. As already indicated in the discussion of section 3.3, it is expected that disk performance will be a major bottleneck of the system. Therefore, it is worthwhile to measure the underlying drives' performance in detail.

The performance of the hard drives is determined in two ways: by the quasi-standard Linux I/O benchmark program bonnie++ (version 1.02a) as well as by directly accessing the block device in streaming mode using the dd(1) data dump program. For the latter, timing and load measurements were performed using the monitoring classes of the MLUC library [173], which rely on the kernel's output under `/proc`. Unless otherwise stated, the load measurements are always given for *both* processors of the test setup nodes, i.e. a load of 100% corresponds to two fully busy CPUs.

As bonnie++ is a program to benchmark hard drives with file systems installed, the test device was

Sequential Output						**Sequential Input**				**Random**	
Per Char		Block		Rewrite		Per Char		Block		Seeks	
K/sec	%CP	K/sec	%CP	K/sec	%CP	K/sec	%CP	K/sec	%CP	/sec	%CP
9961	95	28778	24	12039	11	9563	90	28313	15	195.1	1

Sequential Create						**Random Create**					
Create		Read		Delete		Create		Read		Delete	
/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP
826	99	++++	++++	++++	++++	907	100	++++	++++	2713	100

TABLE 6.1.: Bonnie++ measurement of the test rig disk with an ext2 file system. The sustained write performance is about 29 MB/s and the sustained read performance is about 28 MB/s. Crosses (+) denote that the benchmark could get no accurate values for a particular measurement, since the time for the measurement was too short. Unfortunately, the benchmark did not provide options to adapt the time of these particular measurements.

Hard Disk Performance (Streaming Mode)			
Operation	Throughput	CPU Load	Latency
Read	(28.8 ± 0.5) MB/s	(12 ± 2)%	(4.72 ± 0.02) ms
Write	(27.9 ± 0.5) MB/s	(16 ± 2)%	

TABLE 6.2.: Streaming performance and latency of the test rig disk for direct device access. The measurement was done by dd(1) accessing 1 gigabyte of data in 1 kilobytes blocks. Note that the load is given for two processors, i.e. it is 12% and 16%, respectively, on *each* of the CPUs.

formatted with the Second Extended File System (ext2) as the native Linux file system. The output of a bonnie++ run on the test system is depicted in table 6.1. The streaming performance of the disk is about 29 MB/s for writing and 28 MB/s for reading. Repeated runs on the various test machines showed that the errors in these measurements are around 5%. However, write rates of up to 41 MB/s have also been reported by bonnie++.

The measurements for direct device access, i.e. without an intermediary file system, are shown in table 6.2.[1] For read and write accesses the streaming performance is about 29 MB/s and 28 MB/s respectively, which is in good accordance with the bonnie++ measurements. Due to the larger statistical spread of the bonnie++ measurements and the fact that measurements with dd(1) to a mounted file system delivered similar results compared to direct device access, the values obtained from the direct access are considered to be more trustworthy – particularly as dd(1) is a much simpler program than bonnie++. However, since this benchmark is widely adopted, the bonnie++ values will be stated throughout this section as a comparison. For instance, the latency values provided in table 6.2 and in the tables of the following sections are all derived from the seek times as measured with bonnie++. Although consistently used for all latency measurements throughout this document, it should be noted that the numbers given for latency may not be a very precise measure for the underlying device's seek times for several reasons. The benchmark does not use the whole disk for testing, but

---

[1] All measurements presented in this section have been performed several times and the errors of the respective measurements are the deviations of the corresponding mean value.

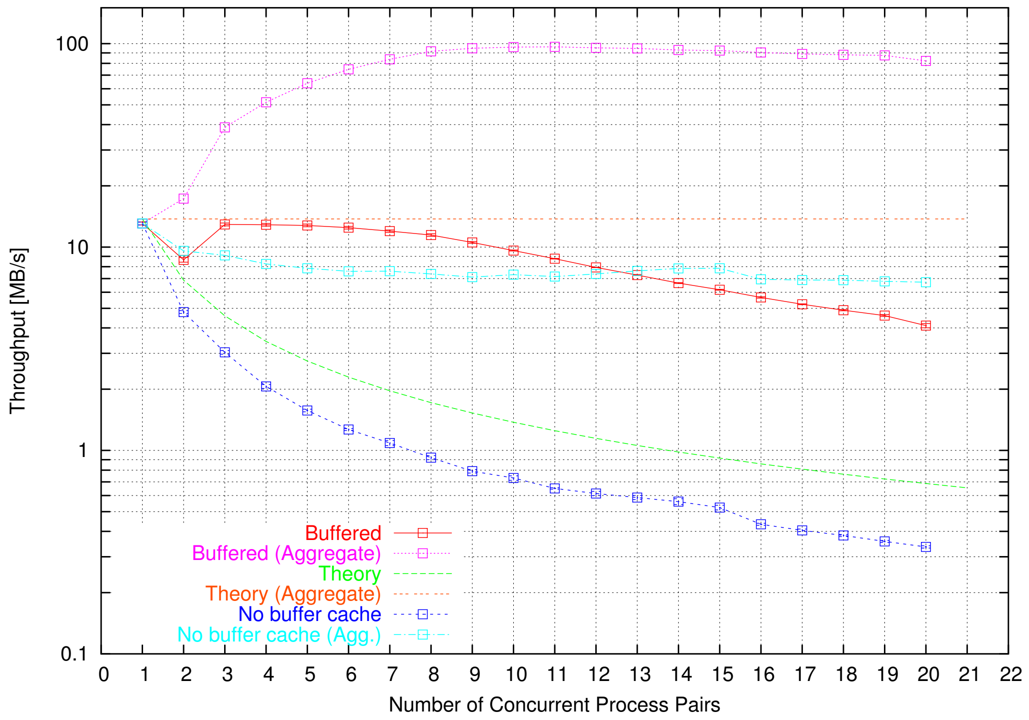

FIGURE 6.1.: Disk throughput vs. concurrent process pairs. All graphs denote the bandwidth available to a single process. The solid red graph provides a measurement with buffer caching, the dashed green line denotes the theoretical reduction of the bandwidth by multiple processes, and the blue graph shows a measurement which bypasses the buffer cache using the DWARW device. In addition, the aggregate bandwidths are provided. Since the measurement was intended to investigate the effect of a preceeding read for every write access, the aggregate bandwidths given take only one process of each pair into account.

only a partition. Hence, the accesses during testing will not span the whole range, but only a fraction. In addition, the benchmark uses three processes that seek on the test partition simultaneously and the benchmark results have proven to be sensitive to the amount of available main memory. Despite these provisos, the results may be helpful in order to compare the *relative* increase of access latency for the various tests.

The performance of the disk when being concurrently accessed by multiple processes is of particular interest, as writes to data storage areas and redundancy updates require read-write or read-modify-write cycles, respectively. Old data must be read – and for redundancy devices XOR'ed with the new data – before a write can be issued. Figure 6.1 shows the disk throughput for process pairs that access the device simultaneously. In each pair, one of the processes is reading from the drive, while the other is writing. In a simplistic model, the disk bandwidth available to single processes should be reduced by a factor of two for every additional process pair. This theoretical limit is indicated by the dashed green curve. However, the buffer cache improves the access bandwidth (red graph) beyond this value. Although 40 processes access the device concurrently, the bandwidth as seen by each of the processes is about 4 MB/s, i.e. 15% instead of 2.5% of the nominal disk throughput. However, as detailed in section 5.4.3, redundancy-side caching would destroy data consistency due to the overwriting of different redundancy updates. Therefore, a measurement which bypasses the buffer cache has been performed. This buffer bypass was realized using a small additional kernel driver, i.e. the DWARW module, which is detailed in Appendix A.3. This measurement is denoted by the dotted blue line in figure 6.1. Obviously, the available throughput is below the limit as given by the above model. This is due to the fact that in the model the bandwidth is simply reduced by a factor of two for every additional process pair. Additional delays, such as the increased overhead for seeking, are not taken into account.

The drop at two process pairs in the graphs with enabled buffer caching is not fully understood. As there are four processes for two processors, the scheduling algorithm was suspected to exhibit an unfair distribution of processor time for this particular combination. However, for some simple test programs neither the number of context switches nor the amount of processor time assigned to the individual processes revealed anomalies that would confirm this assumption. As the measured value is very close to the theoretical curve, the particular characteristics of the components involved may led to buffer cache thrashing. It is not clear, however, why such drops do not occur for larger numbers of process pairs. Furthermore, this irregularity only occurs on SMP kernels – for uni-processor (UP) kernels the graph is smooth. A comparison of this benchmark for SMP vs. UP kernels is shown in figure 6.2.

The load for these write throughput measurements is shown in figure 6.3. While the load is almost a constant for the unbuffered writes, twenty concurrently writing process pairs also render the CPU a system bottleneck for the case of an activated buffer cache.

6.1.2. The Network

Aside from the disk drives, the network is the second main hardware building block in the ClusterRAID's data path. As set out in section 3.3, its performance of course influences the performance of the ClusterRAID. The network bandwidth limits the ClusterRAID throughput and the network latency adds to the disk latency. In order to assess the influence of the underlying network, this section details some performance numbers as measured on the test rig. The aim of the measurements, however, was not to research Ethernet itself, but to understand the performance of ClusterRAID prototype implementation.

Instead of using available network benchmarks, such as Netperf [174], a light-weight tool, called *netIO*, has been developed, which allows for better control of the performed tests. In addition, netIO can use information provided by the previously mentioned Precise Accounting Kernel Patch for more accurate load measurements. Details of the tool can be found in Appendix A.4.

The throughput and load measurements vs. the transferred block size are depicted in figures 6.4 and 6.5 for Fast and Gigabit Ethernet, respectively. Except for the very small block sizes below 20 bytes, the throughput for the Fast Ethernet cards is about 11.2 MB/s. With increasing block size, the ratio of packet overhead to payload decreases at first, as does the load. As soon as the block size reaches the size of the maximum transfer unit, the ratio stays constant and the processor load levels off at about 8% on the sender and about 12% on the receiver. As expected, the load on the receiver is systematically higher than that on the sender due to the interrupt and buffer handling.

The spectrum for the throughput and load measurements of the Gigabit Ethernet cards is more intricate – as typical for Gigabit Ethernet interfaces – and not fully resolved. Although the overall

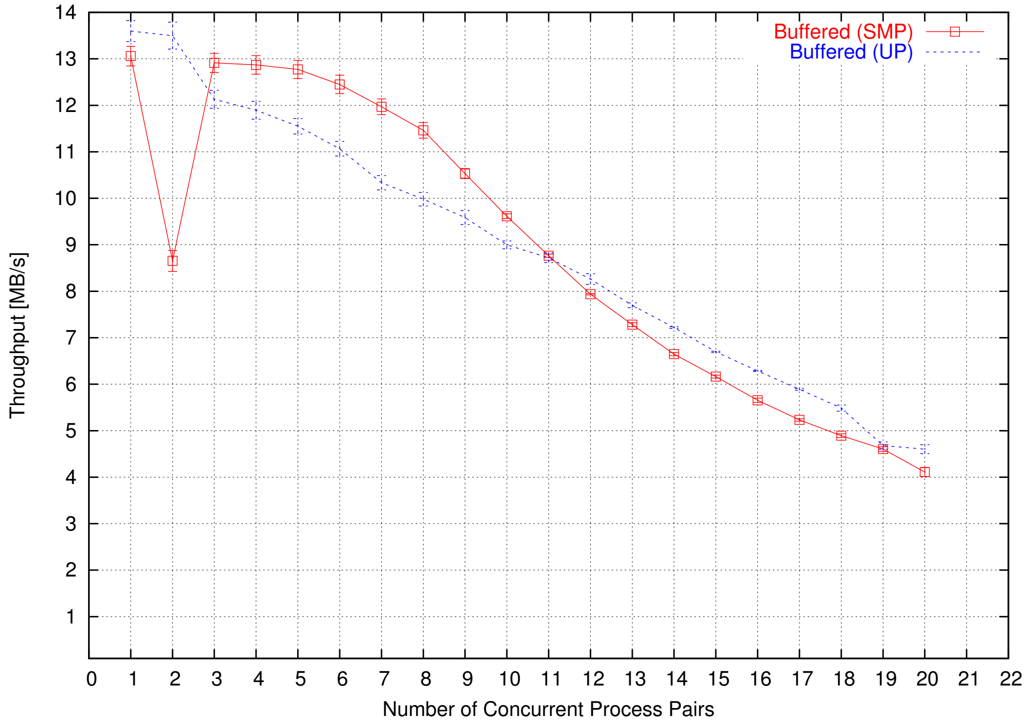

FIGURE 6.2.: Buffered disk throughput for concurrent process pairs on UP (blue) and SMP (red) kernels.

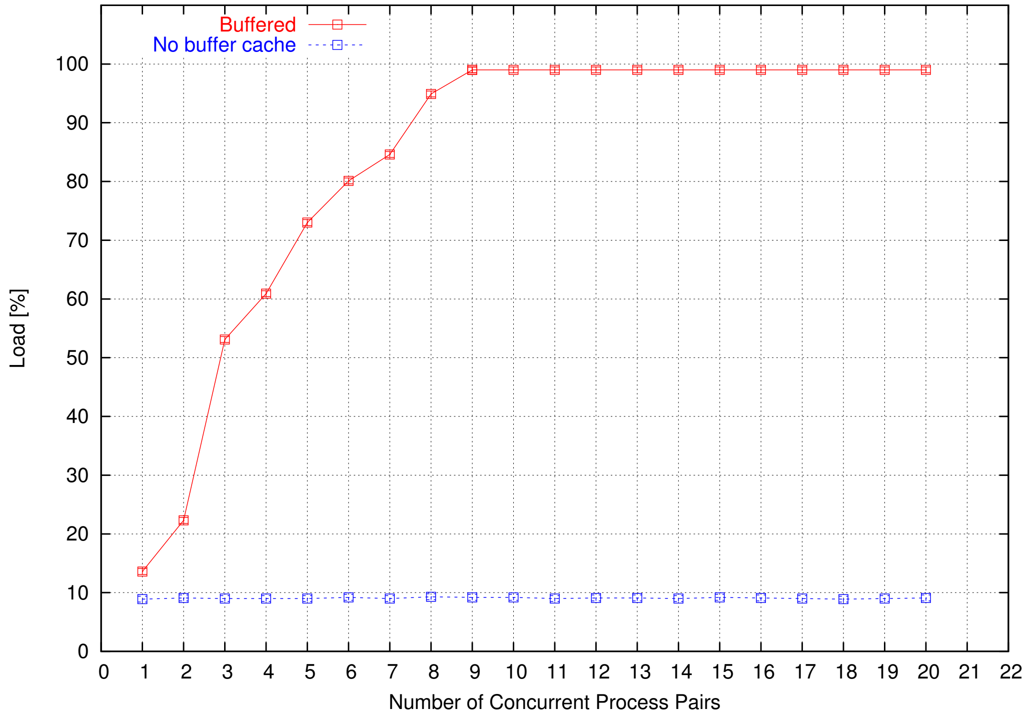

FIGURE 6.3.: CPU load for buffered (red) and unbuffered (blue) disk access by concurrent process pairs.

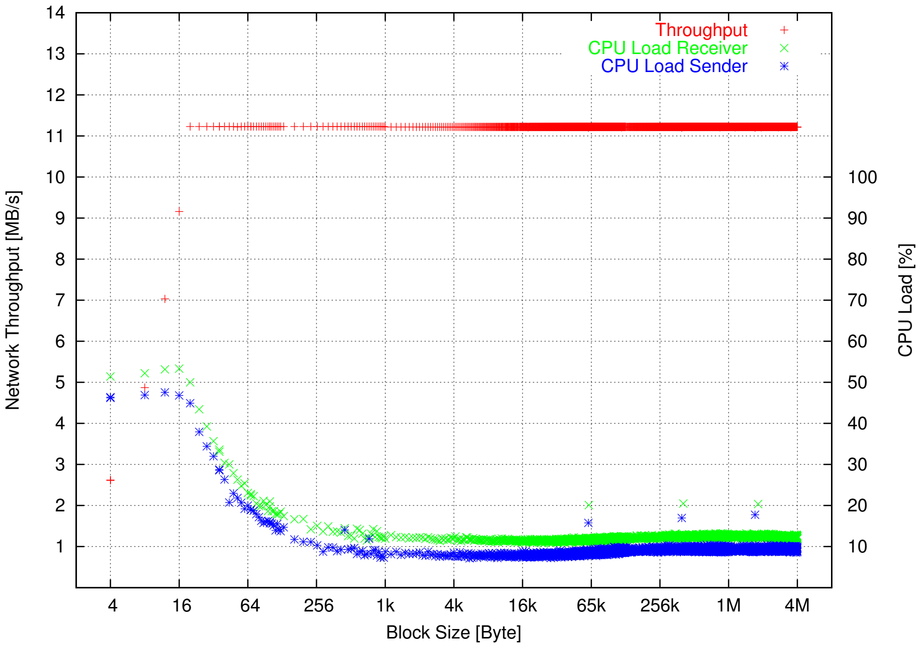

FIGURE 6.4.: Throughput and load benchmark of the Fast Ethernet cards used in the test rig. Repeated measurements showed that the outliers of the load on both sender and receiver are not reproducible. The timely correlation of the load increase indicates that the additional load is induced by processes that are executed on both nodes at almost the same time, such as maintenance activities controlled by the cron daemon.

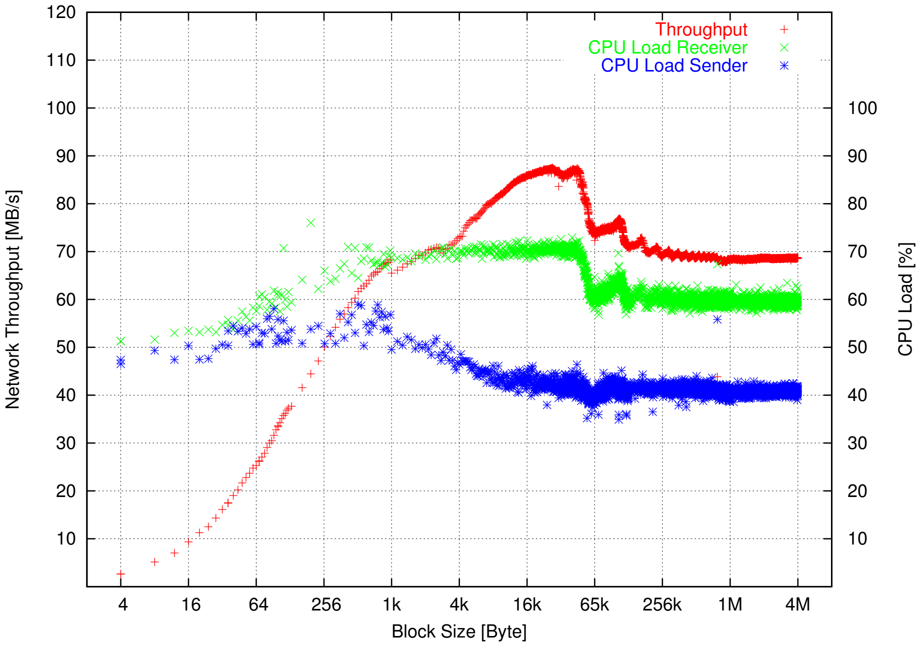

FIGURE 6.5.: Throughput and load benchmark of the Gigabit Ethernet cards used in the test rig.

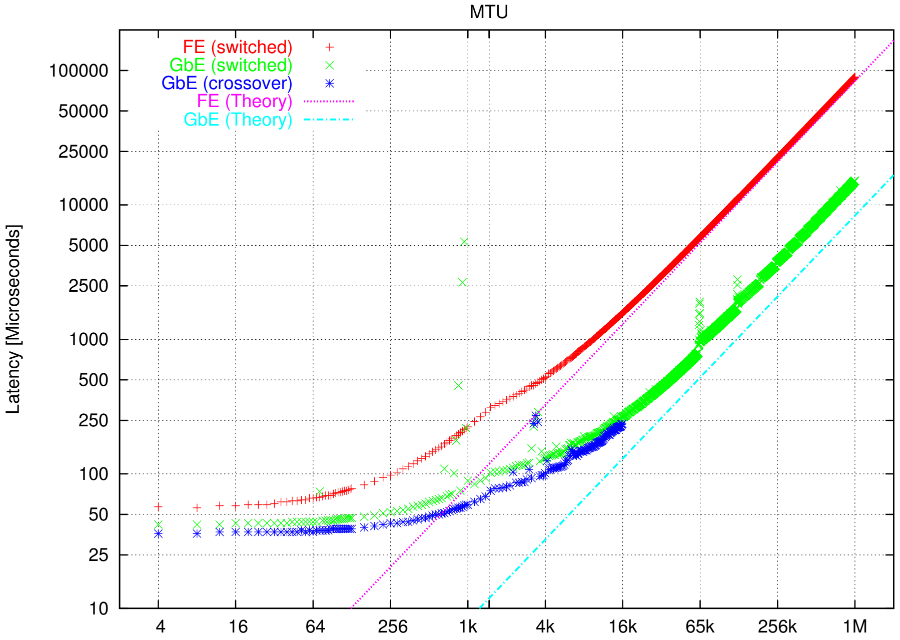

FIGURE 6.6.: Latency measurements of the network cards used: switched Fast Ethernet (red), switched Gigabit Ethernet (green), and crossover Gigabit Ethernet (blue). The crossover measurement is shown only up to block sizes of 16 kilobytes for reasons of clarity, as it overlaps with the switched measurement for larger blocks. The theoretical graphs are derived from ideal networks with bandwidths of 12 MB/s and 120 MB/s for Fast Ethernet and Gigabit Ethernet, respectively.

structure of the graph is typical compared to other measurements with Gigabit Ethernet cards [175], the finer details of the spectra, for instance the non-monotonic behaviour of throughput and load between block sizes of 16 kilobyte and 1 megabyte, are believed to be card, driver, and parameter specific.[2] Therefore, it should only be noted here that the throughput varies between 65 and 87 MB/s in the range of block sizes relevant for the ClusterRAID, i.e. 1 kilobyte and beyond. Block sizes of 1 kilobyte are of special interest, since the ClusterRAID driver states this size as its preferred request size, when registering with the core kernel. If multiple requests are merged by the kernel, multiples of this block size are transferred. The CPU load in this regime is about 40-50% on the sender and about 60-70% on the receiver.

The dependency of the latency on the transferred block size for the two cards are shown in figure 6.6. Values for block sizes of 1 kilobyte range between 100 and 250 microseconds for Gigabit and Fast Ethernet, respectively. The slopes of all three graphs exhibit discontinuities at the standard Ethernet maximum transfer unit (MTU) at 1500 bytes, where packets are split up before transmission. The latency measurements of the switched Gigabit Ethernet shows heavy fluctuations at packets sizes just below 1 kilobyte, also in repeated measurements. Since this structure does not occur with the crossover connection, it is accounted to a not fully understood effect of the switch. The outliers in the range of 4 kilobyte packets are not reproducible. The latency increase at packet sizes of 65 kilobytes and 128 kilobytes in the switched Gigabit Ethernet graph are ascribed to buffer size boundaries. Compared to the disk seek times, which are in the range of several milliseconds, the network will contribute minimally to the ClusterRAID access latency. The contribution of a switch in the network path to the total latency is at the level of 10 microseconds.

---

[2] For all measurements, the default network options have been used. Therefore, the results may differ if parameters, such as the socket buffer sizes or the TCP_NODELAY option, are changed. A very detailed measurement of the network characteristics observed when varying these parameters can be found in [176].

6.1.3. The Network Block Device

The network block device is the main external software building block of the ClusterRAID architecture. It relies on the two hardware components examined in the previous sections, disks and network. For all measurements presented in this chapter, the NBD implementation as shipped with the standard Linux kernel is used. However, as the ClusterRAID only relies on the block device abstraction, the NBD could be replaced by other systems, such as iSCSI devices or the Enhanced NBD, which has been introduced in section 2.4.1. Although the kernel NBD does not have as many features as the Enhanced NBD, for instance, it has been chosen due to its simpler setup. In production systems at least some of the advanced features of the ENBD, such as the automatic reconnect after reboots, will be indispensable.

The results of the NBD measurements using dd(1) are summarized in tables 6.3 and 6.4. The former contains the measured values for a point-to-point NBD connection with default parameters. In the latter, the server accesses the underlying resource with synchronous file I/O. As set out in section 5.4.3, this is necessary in order to avoid data corruption.

The throughput of the NBD is mainly limited by the performance of the underlying hardware, i.e. disks and network. In default mode, the Fast Ethernet network limits the throughput to about 9 MB/s for reads and 11 MB/s for writes. If Gigabit Ethernet is used, the throughput is about 28 MB/s for both read and write accesses in streaming mode. These measured values are even slightly higher than the results of the locally tested disk. This is accounted for by the doubling of available memory for the buffer cache, as there are now two computers involved in the data transfer. As expected, the read performance is unaffected by whether the underlying resource on the server being opened in synchronous or asynchronous mode. However, the write throughput is degraded from 28 MB/s to about 18 MB/s. Due to the correlation of throughput and load, the server-side load is reduced from 40% to about 23%.

The access latency for current disks is in the range of 5 to 10 ms, whereas the network latency – even for low-cost technologies such as Ethernet – is in the range of a few 100 μs at most. Hence, the latency increase by the network is expected to be small. This is confirmed by the latency measurements in tables 6.3 and 6.4. The increase is below 5 percent for Fast Ethernet, and a little higher for Gigabit Ethernet, in contrast to the network latency measurements in the previous section.

As network block devices constitute a fundamental building block of the ClusterRAID architecture, and the prototype implementation in particular, the results of tables 6.2 and 6.4 define the boundaries for the ClusterRAID performance measurements in the upcoming sections.

6.2. ClusterRAID Performance

The performance numbers for the underlying hardware given in the previous section set the limit for the maximum achievable performance of the ClusterRAID prototype. In this section, the individual components of the system will be analyzed using a bottom-up approach. Initial tests investigate the

NBD Default Mode				
Fast Ethernet				
Operation	Throughput	Client CPU Load	Server CPU Load	Latency
Read Write	(8.9 ± 0.1) MB/s (10.9 ± 0.01) MB/s	(13 ± 2)% (17 ± 2)%	(10 ± 2)% (17 ± 2)%	(4.89 ± 0.01) ms
Gigabit Ethernet				
Operation	Throughput	Client CPU Load	Server CPU Load	Latency
Read Write	(27.9 ± 0.01) MB/s (28.7 ± 0.06) MB/s	(42 ± 2)% (30 ± 2)%	(35 ± 2)% (40 ± 2)%	(5.17 ± 0.02) ms

TABLE 6.3.: NBD performance measurements for the default setting.

NBD Synchronous Mode				
Fast Ethernet				
Operation	Throughput	Client CPU Load	Server CPU Load	Latency
Read Write	(8.9 ± 0.1) MB/s (8.1 ± 0.1) MB/s	(13 ± 2)% (15 ± 2)%	(10 ± 2)% (15 ± 2)%	(4.94 ± 0.05) ms
Gigabit Ethernet				
Operation	Throughput	Client CPU Load	Server CPU Load	Latency
Read Write	(27.9 ± 0.02) MB/s (17.6 ± 0.01) MB/s	(42 ± 2)% (25 ± 2)%	(35 ± 2)% (23 ± 2)%	(5.05 ± 0.01) ms

TABLE 6.4.: NBD performance measurements in synchronous mode.

performance of the system for local access only, i.e. with no remote devices attached. These tests are followed by those where a network block device is part of the data path. Finally, the performance results of a fully assembled ClusterRAID are presented. This scheme allows one to identify the major restrictions, and to understand how the constituent parts contribute to the performance of the whole system. This information could also form the basis for future improvements to the ClusterRAID implementation.

6.2.1. Local Testing

The local performance of a ClusterRAID device is summarized in table 6.5. Basically, there are two operations, read and write, on two types of devices, i.e. data and redundancy devices. Reading data from a ClusterRAID device is mainly limited by the performance of the underlying hard disk. The effect of the additional ClusterRAID layer in the data path is measurable but small. The ClusterRAID's read throughput of 27.6 MB/s is about 96% of the measured hard disk rate. The load on the CPU is insignificantly increased during reads when compared to a direct disk access. As the device type does not have a fundamental impact on read accesses, the performance measurements for reading from a redundancy device yield approximately the same values. Reading from a redundancy device only occurs during data reconstruction.

ClusterRAID Local Access

Operation	Throughput	CPU Load	Latency
Data Read	(27.6 ± 0.3) MB/s	(14 ± 2)%	(6.1 ± 0.1) ms
Data Write	(11.0 ± 0.2) MB/s	(19 ± 2)%	
Data Write (CM)	(10.6 ± 0.2) MB/s	(25 ± 2)%	(4.7 ± 0.6) ms
Redundancy Read	(27.7 ± 0.7) MB/s	(14 ± 2)%	(no cache)
Redundancy Write	(11.7 ± 0.3) MB/s	(19 ± 2)%	

TABLE 6.5.: ClusterRAID performance measurements for local access. The write in `COPYMODE` is indicated by a CM in brackets.

Writing data to a ClusterRAID device is always preceeded by a read. This holds true for both data and redundancy devices. As the disk measurements with concurrently reading and writing processes have shown, this simultaneous access leads to a significant reduction in available bandwidth. For two processes the disk transfer rate was reduced from around 28 MB/s to 13 MB/s. The ClusterRAID write performance of around 11 MB/s is slightly lower for two reasons. Firstly, for the disk measurement, the two processes were completely independent. Hence, to a certain degree the kernel was able to arrange the disk accesses in the way it considered them to be optimal. In the ClusterRAID, the read and write accesses to specific blocks are closely correlated. After reading a specific block, and possibly the consecutive ones, the disk head must move back in order to complete the write. This results in a back-and-forth movement of the disk head, having a negative impact on the performance. Secondly, the two independent processes in the case of the disk only measurement benefit from the buffer cache. If the writing process accesses a block before the reading process does, the latter can retrieve the data from the cache. However, since the resulting bandwidth is approximately reduced by a factor of two, this does not occur frequently. The ClusterRAID logically operates below the buffer cache and cannot profit from such coincidences.

A difference in the data path between the writing of data or redundancy information to the underlying device is that in the latter case the read data is directly XOR'ed with the content of the request buffer, while in the former case the original content is forwarded unchanged to the underlying data device. On the other hand, the corresponding blocks of the data device must be locked to prevent data corruption before the driver can process the requests, which is not necessary for a redundancy device. This additional overhead explains the slightly worse write performance of a data device compared to a redundancy device.

As outlined in section 5.4.2, the ClusterRAID must be able to copy the data of each incoming request in order to avoid data corruption by later external access. Therefore, an additional measurement with enabled copying (`COPYMODE`) was performed. Although the throughput is almost unaffected, the additional copy step increases the processor load from 19% to 25%. For a redundancy device this copy step is not required.

The results for contiguous access as obtained using the bonnie++ benchmark are in good accordance with the results from the direct access measurements just described, cf. table 6.6. The access latency through the ClusterRAID layer to the constituent disk drive is slightly increased compared to the value measured for direct access. However, as mentioned in the section concerning hard drive

Sequential Output						**Sequential Input**				**Random**	
Per Char		Block		Rewrite		Per Char		Block		Seeks	
K/sec	%CP	K/sec	%CP	K/sec	%CP	K/sec	%CP	K/sec	%CP	/sec	%CP
8539	89	13236	13	6831	7	9076	86	27630	17	153.2	1

Sequential Create						**Random Create**					
Create		Read		Delete		Create		Read		Delete	
/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP	/sec	%CP
783	99	++++	++++	++++	++++	904	99	++++	++++	2695	99

TABLE 6.6.: Bonnie++ measurement of a ClusterRAID device in COPYMODE, but with no attached redundancy devices. Crosses (+) denote that the benchmark could get no accurate value for a particular measurement.

performance, this measurement is obviously sensitive to the amount of available main memory. If the ClusterRAID uses a private cache that is aggressively allocated during module loading, the seek times measured are increased by around 20% for both tests, with and without access through the ClusterRAID device.

The write bandwidth to a local ClusterRAID device when being accessed concurrently by multiple processes is depicted in figure 6.7. As for the disk only measurement, the performance has been measured with and without the buffer cache, denoted by the red and the blue graph, respectively. If the number of concurrent processes is increased, the available write bandwidth for each process is reduced due to the serialized access. It should be noted, however, that this measurement represents the worst case scenario, and that preventive measures, such as request coalescing, are not taken into account.

6.2.2. Remote Testing

Data devices are not only accessed from the node they reside on, but also from remote nodes. The data reconstruction in case of a device failure, the replacement of a node by a spare one and the subsequent reconstruction of its data, or the recalculation of the information of a redundancy device are situations in which reading blocks from a remote data device becomes necessary. In the first two scenarios, additional blocks must be read from remote redundancy devices. There is no scenario where a node writes blocks to a remote data device. Writing to a remote redundancy device, however, is necessary for every change of local data.

Table 6.7 lists the benchmark results of the measurements for these operations. As for local access, the read performance does not differ between data and redundancy devices. The throughput is limited by the network block device or the underlying hardware, respectively. For Fast Ethernet the network limits the read speed, while for Gigabit Ethernet the remote hard drive limits the throughput. The numbers are very close to the results obtained by the network block device measurements of section 6.1.3.

For writes, the throughput is limited by the transfer rate of the redundancy device. The measurements, 11.0 MB/s for Fast Ethernet and 12.7 MB/s for Gigabit Ethernet, indicate that the available network bandwidth also influences the throughput. Since the ClusterRAID prototype implements no lazy update schemes, data integrity requires the synchronization of all requests with the underly-

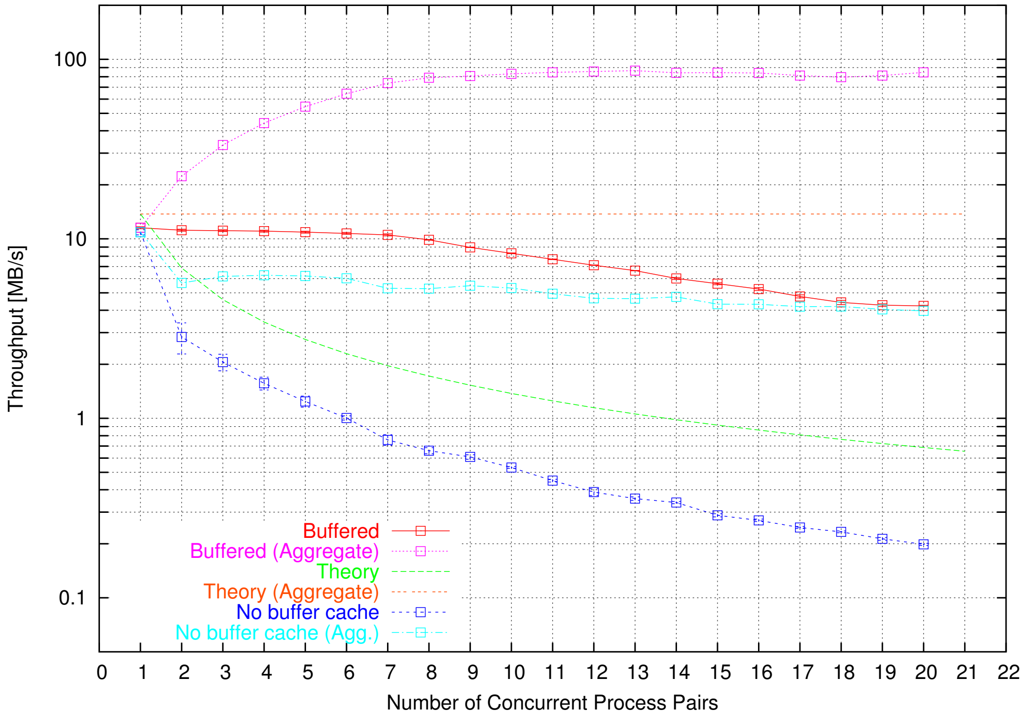

FIGURE 6.7.: ClusterRAID write throughput vs. the number of accessing processes. All graphs denote the bandwidth available to a single process. The red graph is a measurement with buffer caching, the dashed green line denotes the theoretical reduction of the bandwidth by multiple processes, and the blue graph shows a measurement which bypasses the buffer cache using the DWARW device. As in figure 6.1, the aggregate bandwidths are provided in addition.

ing devices. Therfore, additional measurements with the NBD opening the resource in synchronous mode have been performed. The decrease of the bandwidth to 6 MB/s and about 8 MB/s for Fast Ethernet and Gigabit Ethernet, respectively, underline that the required synchronicity, as currently implemented, has a significant impact on the ClusterRAID's write performance. Although the NBD can write at around 17 MB/s in synchronous mode, the combination with the redundancy device throttles the performance at this point.

6.2.3. System Tests

Having examined the performance of the underlying building blocks of the ClusterRAID, this next section focuses on the evaluation of a fully assembled ClusterRAID. All measurements in this section were performed on an eight node subcluster using Gigabit Ethernet as the interconnect. Unless explicitely stated otherwise, all measurements include the locking, copy, and synchronization overhead necessary for data consistency. Hence, the measured values reflect realistic values of the ClusterRAID prototype implementation. Throughout the remaining chapter the notation (N,P) refers to a setup with N nodes in total, of which P nodes are dedicated to the storage of redundancy information.

ClusterRAID Remote Access				
Fast Ethernet				
Remote Operation	Throughput	Client CPU Load	Server CPU Load	Access Latency
Data Read	(8.8 ± 0.1) MB/s	(13 ± 1)%	(10 ± 1)%	(5.54 ± 0.01) ms
Red. Read	(8.9 ± 0.1) MB/s	(13 ± 1)%	(11 ± 1)%	
Red. Write	(11.0 ± 0.01) MB/s	(30 ± 1)%	(13 ± 1)%	(4.96 ± 0.07) ms
Red. Write (sync)	(6.0 ± 0.01) MB/s	(14 ± 1)%	(10 ± 1)%	(no cache)
Gigabit Ethernet				
Remote Operation	Throughput	Client CPU Load	Server CPU Load	Access Latency
Data Read	(27.8 ± 0.06) MB/s	(41 ± 2)%	(36 ± 2)%	(5.53 ± 0.01) ms
Red. Read	(27.8 ± 0.02) MB/s	(43 ± 2)%	(37 ± 1)%	
Red. Write	(12.7 ± 0.36) MB/s	(16 ± 4)%	(30 ± 1)%	(5.14 ± 0.03) ms
Red. Write (sync)	(7.9 ± 0.01) MB/s	(10 ± 2)%	(17 ± 2)%	(no cache)

TABLE 6.7.: ClusterRAID performance for remote access.

Read

For successful reads, the ClusterRAID behaves as a set of completely independent nodes. Hence, the aggregate read throughput scales perfectly with the number of data nodes, and the throughput per node is independent of the number of redundancy or data nodes. This behaviour is shown in figure 6.8. The measurement for local accesses as summarized in table 6.5 on page 130 indicated that the overhead imposed by the ClusterRAID for successful reads is very small when compared to direct disk accesses. Since successful read requests are served from the local data device, no network traffic is required and, thus, no additional CPU or network load is generated.

Write

In case of data writes, redundancy information is stored remotely on the corresponding redundancy areas. As discussed in section 5.4.3, the prototype implementation relies on synchronous redundancy writes to ensure data consistency. Table 6.7 indicates that for remote accesses to a redundancy area this approach limits the effective throughput to about 8 MB/s. In the actual implementation this bandwidth is shared by concurrently writing data nodes. The effect of competing data nodes on the bandwidth available to a single node and the corresponding processor load is depicted in figures 6.9 through 6.12 for different setups, (8,1) and (8,2).

As expected, the bandwidth available to a single data node decreases with the number of concurrent writers. This behaviour is analogous to that of multiple processes all of which are accessing the same device, see figure 6.1 or 6.7. The maximum transfer rate is around 6.6 MB/s for an (8,1) configuration with a single writing process. The reduction from 8 MB/s to 6.6 MB/s is due to the additional read on the data node and the enforced synchronicity with the redundancy update. Obviously, this value is independent of the total number of nodes and the same for all (N,1) setups. For

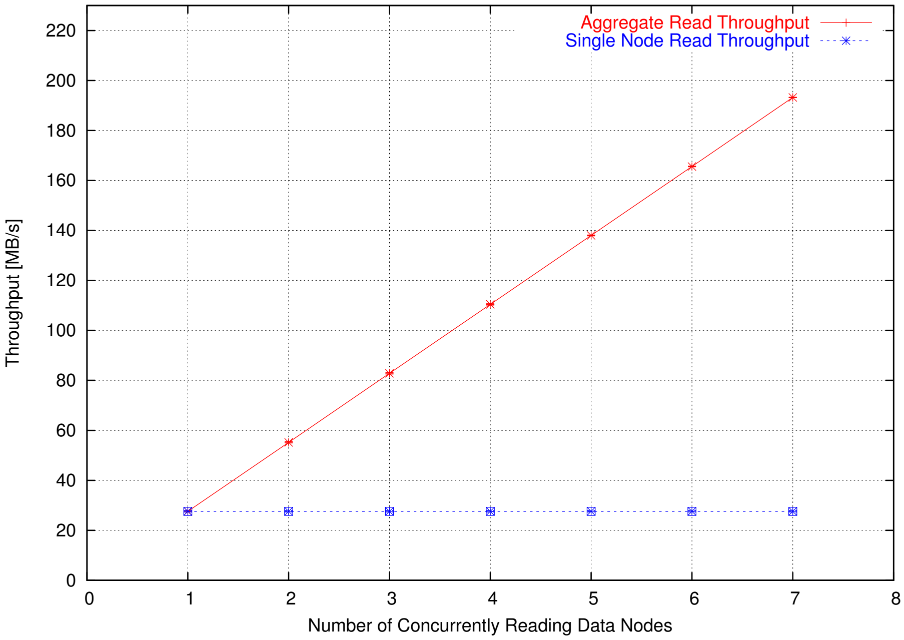

FIGURE 6.8.: ClusterRAID read throughput for an (8,1) setup.

the (8,1) setup, the aggregate throughput of all nodes is almost a constant and levels off at around 6 MB/s. The constant throughput is reflected in an almost constant load on the redundancy node, see figure 6.10, whereas – due to the correlation of throughput and load – the load on the data node decreases when more writers are added. The throughput and load for an (8,2) setup are depicted in figures 6.11 and 6.12, respectively. Obviously, the required update of an additional redundancy node has a strong impact on the throughput. For a single writer the throughput is reduced to 3.7 MB/s and the aggregate throughput is reduced to about 5 MB/s. The load values are comparable to the ones in the (8,1) setup.

The impact of a variable number of redundancy storage areas on a data node's CPU load and its available bandwidth for a single writer is shown in figure 6.13. As expected, the write throughput decreases with the number of redundancy devices to be updated, since the required synchronization is increased. The significant increase in CPU load is due to both the additional expense computing the redundancy information and the additional overhead for the increased network transfer, as shown in figure 6.14.

In order to identify the main limiting factors for the write throughput, figure 6.15 shows the throughput for a single writer on an (8,2) setup if specific options are disabled. Compared to a UP system, the throughput can be slightly improved on an SMP system. Within the error range the disabling of locks does not improve the throughput. However, this is mainly due to the single writer setup, since there is no competitor for the remote lock. The result may differ if multiple writers are used. The main contribution for throughput improvement, however, can be achieved by opening the redundancy storage in non-synchronous mode. Although the value is not quite as high as for local access, the throughput is improved by a factor of three when compared to the regular access. These results will provide a basis for the discussion of possible improvements in section 6.4.

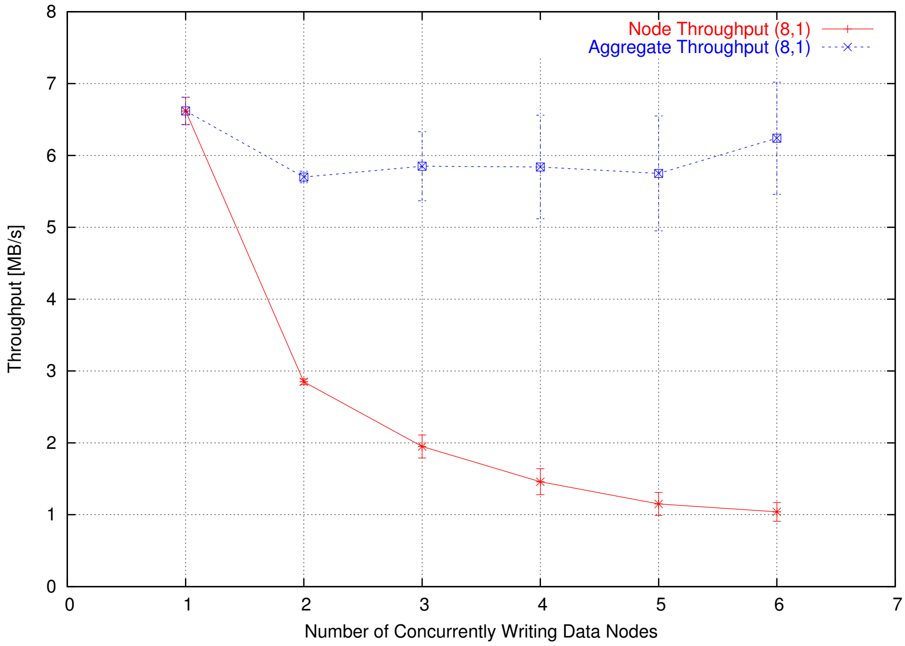

FIGURE 6.9.: ClusterRAID write throughput for an (8,1) setup.

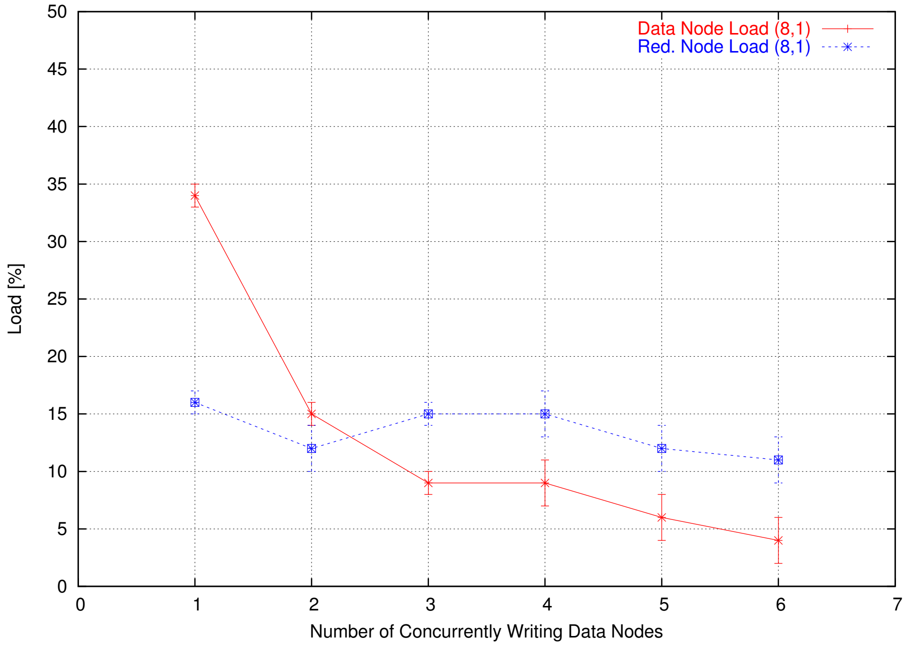

FIGURE 6.10.: ClusterRAID CPU load for writes in an (8,1) setup.

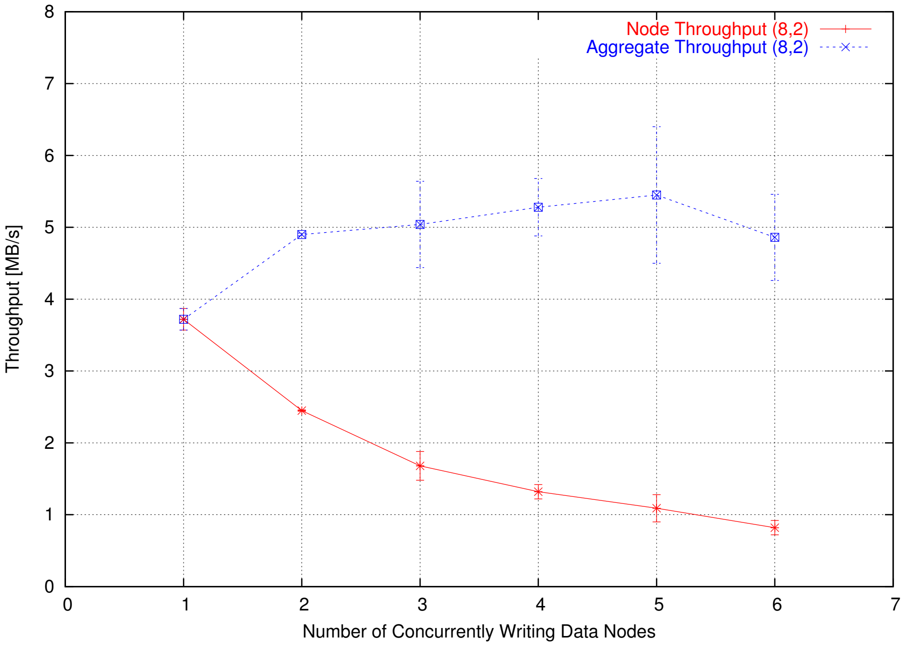

FIGURE 6.11.: ClusterRAID write throughput in an (8,2) setup.

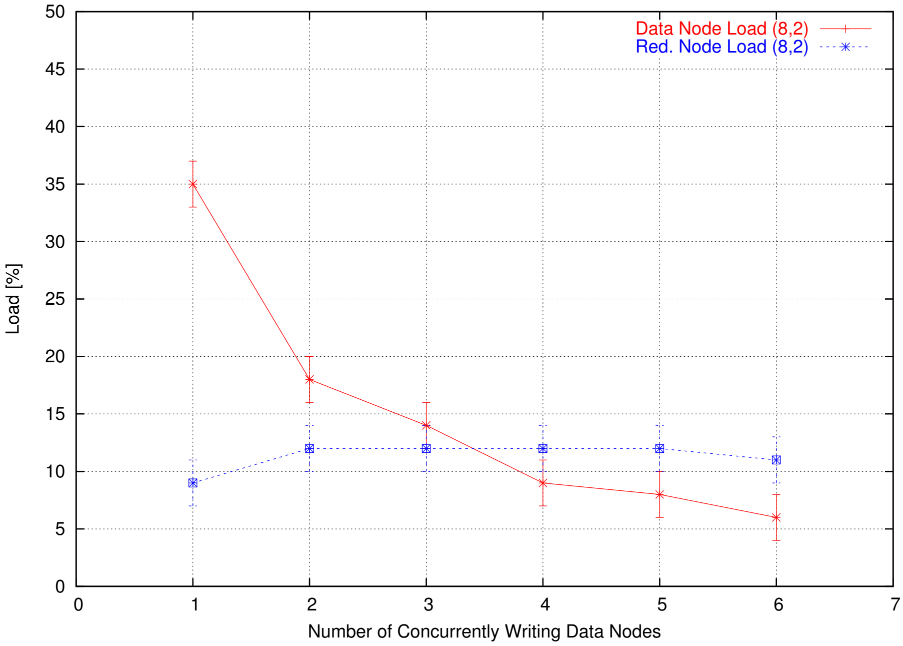

FIGURE 6.12.: ClusterRAID CPU load for writes in an (8,2) setup.

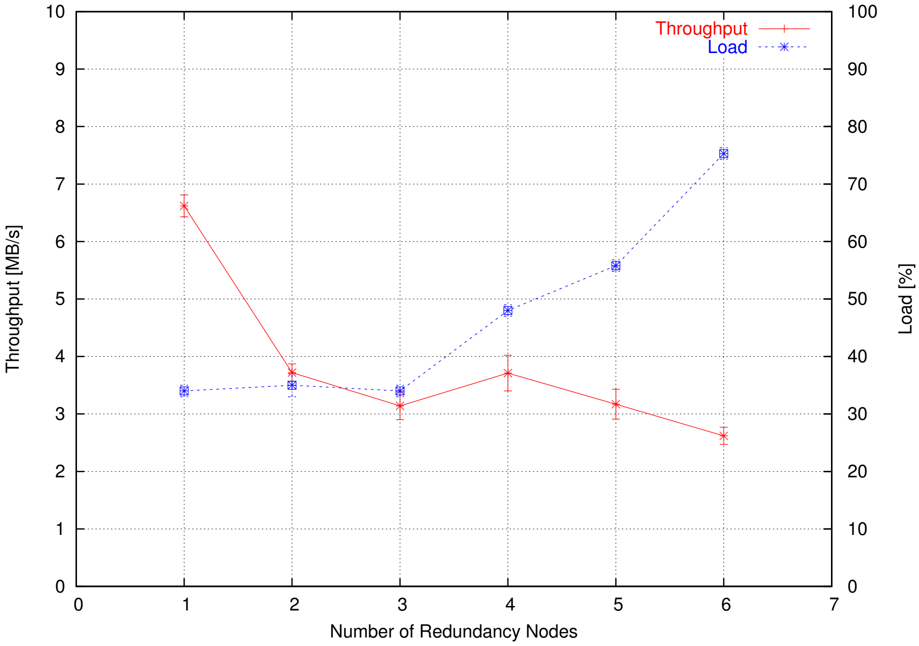

FIGURE 6.13.: Write throughput and CPU load on a data node for a variable number of redundancy nodes. The measured values are obtained for a single writer. The CPU load on the redundancy nodes is independent of the number of redundancy nodes in the setup. Therefore, the numbers given in figure 6.12 also hold for this measurement.

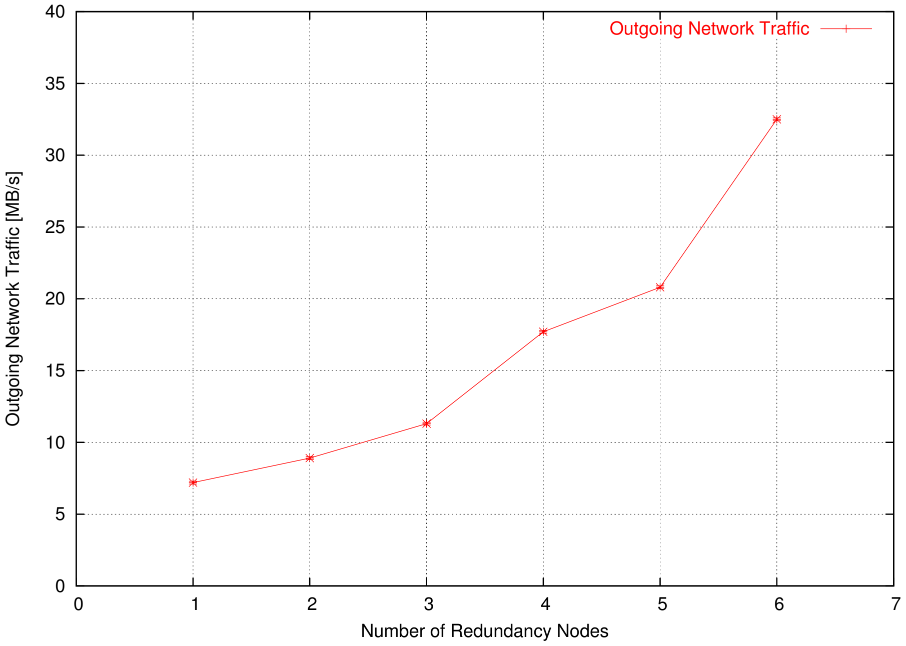

FIGURE 6.14.: Outgoing network traffic on a data node for a variable number of redundancy nodes. The measured values include all packet overhead.

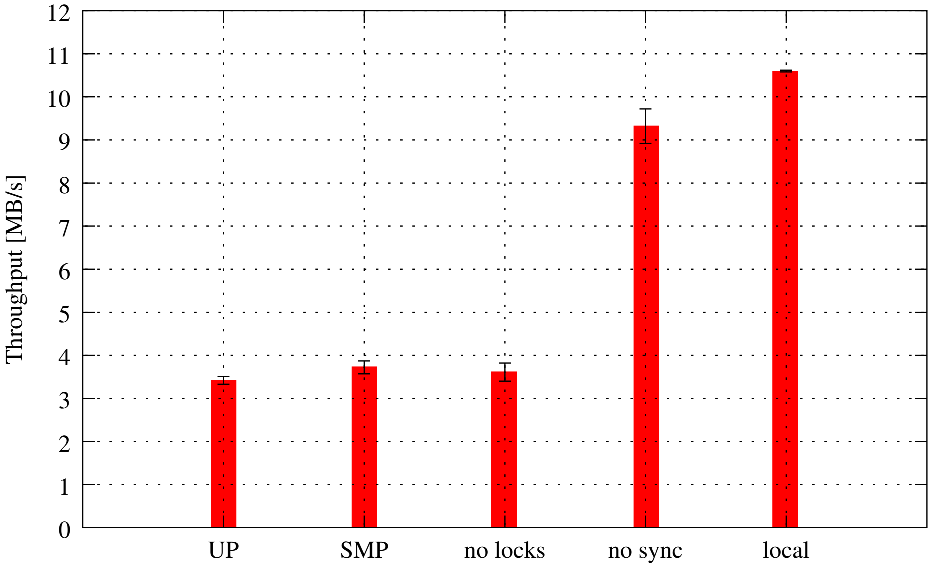

FIGURE 6.15.: Comparison of the contributions to the throughput limitation for a single writer on an (8,2) system. From left to right: single processor kernel, SMP kernel, no remote locking, no synchronous mode for the redundancy storage area, and local access.

Reconstruction

In general, a fast reconstruction is not as essential for the ClusterRAID as it is for other RAID architectures, since the increased vulnerability in degraded mode can be compensated by using a sufficient degree of redundancy. However, working in degraded mode inevitably leads to lower reliability, higher processor and network load, and reduced performance for all nodes involved in the reconstruction. Therefore, it is desirable to also investigate the performance during reconstruction and to understand possible bottlenecks.

The throughput and load as a function of the number of nodes required to reconstruct the data of a faulty device are shown in figure 6.16. For these measurements ClusterRAID setups with an (N,1) configuration have been used, where N is varied between 3 and 8. For the performance it is irrelevant whether the remote data is read from a data node or a redundancy node. From the plot it is obvious that the load induced by the reconstruction is the limiting factor in all configurations. It is always beyond 95%. With an increasing number of nodes the throughput as seen by the application decreases for two reasons. Firstly, when more nodes belong to a redundancy ensemble, more remote data must be transferred in order to reconstruct the local data. The additional network traffic reduces the CPU time available for the actual decoding. Secondly, the effort of decoding increases with the number of nodes per redundancy ensemble. The polynomial complexity depends on the decoding scheme being used. The cost to reconstruct a symbol using the matrix inversion, for instance, is

$$C_{inv} = (N-P)\cdot M + (N-P-1)\cdot A, \tag{6.1}$$

where M and A are the cost for multiplication and addition over the Galois field, respectively. The time complexity scales practically linearly with the (N-P) nodes needed to reconstruct the data. Due

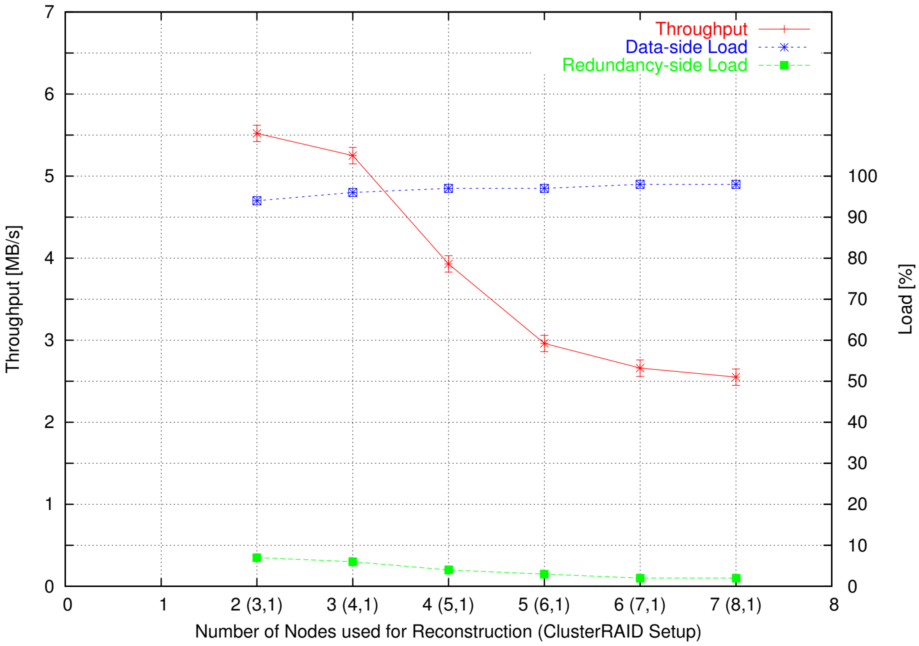

FIGURE 6.16.: Throughput and load during reconstruction as a function of the number of nodes required for reconstruction. The ClusterRAID setup used for the individual measurements is given in brackets at the corresponding ticks of the x-axis. The throughput is the bandwidth at which the application can read the reconstructed data. The data-side load is measured on the reconstructing node, the redundancy-side load is measured on the redundancy node. The CPU load on the other data nodes is comparable to the one on the redundancy node.

to the forward-backward substitution of decoding by means of the LU decomposition, the complexity of the latter approach is quadratic with the number of nodes. Since the cost for a Galois multiplication can be up to two orders of magnitude higher than for a Galois addition (effectively an XOR operation), the multiplication term in equation 6.1 will dominate. A thorough investigation of the scaling of time required for a Galois multiplication with the CPU clock speed has shown that using faster processors may help to speed up the reconstruction [177]. Alternative approaches applied to the prototype implementation and further suggestions to improve the performance will be given in section 6.4.

The load on the redundancy node is below 10% for the regime considered here. It slightly decreases with the number of nodes due to the fact that the relative fraction of data requested from the redundancy storage area decreases as the redundancy ensemble becomes larger.

Table 6.8 shows the results of additional measurements for specific ClusterRAID configurations. For the (8,2) setup, measurements were performed to determine the impact of the number of faulty devices during reconstruction and to study the effect of concurrent reconstruction on multiple data nodes. The results indicate that the throughput per node is slightly improved when two devices are marked faulty instead of one. This is not surprising as the current implementation imports *all available* data for reconstruction, not only the minimum necessary. Unneeded data is simply dropped on the reconstructing node. Since the number of unneeded data blocks is usually small compared to the

ClusterRAID Reconstruction Performance							
Setup	F	R	W	I	Data load	Red. load	Throughput
(8,1)	1	1	0	7	(98±1)%	(2±1)%	(2.55±0.01) MB/s
(8,2)	1	1	0	7	(93±2)%	(2±1)%	(2.29±0.03) MB/s
	2	1	0	6	(92±2)%	(3±1)%	(2.34±0.03) MB/s
	2	2	0	6	(90±1)%	(6.5±1)%	(2.35±0.03) MB/s
(7,1)	1	1	0	6	(98±1)%	(2±1)%	(2.66±0.01) MB/s
	1	1	1	6	(51.5±1)%	(5.5±1)%	(1.38±0.06) MB/s

TABLE 6.8.: ClusterRAID performance during reconstruction. F: number of devices marked faulty, R: number of reconstructing nodes, W: number of writers, I: number of devices involved in reconstruction, Data load: CPU load on the data node, Red. load: CPU load on the redundancy node. The throughput is as seen by a reconstructing application.

total number of blocks in a redundancy ensemble, this dropping of blocks was accepted for the prototype implementation, as it simplified the implementation. Within the error limits, the throughput for one and two reconstructing nodes is the same. This is due to the fact that the data-side CPUs represent the primary limit, and not any server-side component. The doubled load on the server-side supports this conclusion. For the (7,1) setup a measurement with a concurrently writing data node was performed. As outlined in section 5.4.1, the data read by a reconstructing node must be consistent in order to be able to decode the correct data. Therefore, locks of sets of blocks to be decoded are acquired before the data is reconstructed. If another data node is concurrently writing data, i.e. accessing the redundancy storage areas from which the redundancy information is read, this other node competes with the reconstructor for the corresponding locks. The concurrent locking and the additional write accesses to the redundancy device reduce the speed of the reconstruction and the data-side load by a factor of approximately two. The load on the redundancy node, however, increases due to the additional write operations.

A comparison of configurations with the same number of faulty devices and nodes involved, such as the (8,1) setup with the (8,2) setup, is intricate. Although the number of involved nodes is the same, the reconstruction throughput for the (8,1) setup is higher. A part of the data is dropped in the (8,2) setup, and hence the corresponding fraction of the bandwidth is wasted, the decoding in this scenario should require less computing power according to equation 6.1.

6.3. Functional Testing

A number of tests have been performed to verify the basic functionality, stability, and data consistency of the prototype implementation. These tests include the investigation of the prototype's behaviour during failures, its ability to replace a faulty node by a spare one, and its interplay with a distributed file system.

6.3.1. Online Repair and Node Replacement

Figure 6.17 depicts a trace of the ClusterRAID throughput, the network traffic, and the CPU load during a functional test of a (3,1) system. The disk of a ClusterRAID data node is accessed by an application at almost the nominal disk transfer rate (dashed line). The blue graph denotes the variation of this throughput over time as available to the application. After about 20 seconds, the local device is marked as faulty and is no longer available for data access by the application. The online repair mechanism of the ClusterRAID immediately starts to import data and redundancy information from the other nodes to reconstruct the requested data on-line. Due to required accesses to remote devices and the decoding of the data the transfer rate drops to about 5.5 MB/s. As mentioned earlier, the rate is limited by the capabilities of the performance of the decoding CPU. The red graph shows the outgoing network traffic of the redundancy node. Obviously, this rate closely corresponds to the reconstruction bandwidth as seen by the application. After about 40 seconds, the application stops accessing the ClusterRAID device. The corresponding node is removed from the ClusterRAID system and replaced by a spare node. The incoming network traffic of this new node is given by the green graph. Since in the current implementation the reconstruction of data to a spare node is done via the same mechanisms as the online reconstruction, the outgoing network rate of the redundancy node is at a comparable level. Due to the setup with three nodes, the incoming network traffic of the spare node is twice the outgoing rate on the redundancy node. Once the replacement and reconstruction are complete, the ClusterRAID system returns to its ground state (not shown in the figure). The only difference in this final state compared to the initial state is that one of the data nodes has been replaced by a spare node.

The load on the data node, the redundancy node, and the spare node during this test, are depicted also in figure 6.17. The figure shows the striking differences in load during the different phases of the test, i.e. the initial failure of the accessed data device, the replacement, and the start of the resynchronization.

6.3.2. Stability and Consistency

The ClusterRAID prototype has undergone extensive testing in order to verify its stability and its data consistency under long term and/or concurrent accesses. In a (6,2) setup the direct device access has been tested under continuous load for 10 days. During this time, every ClusterRAID driver has transferred more than 1 TB of data without problems. An (8,2) setup with ext2 file systems on every data node has been used under continuous bonnie++ access in a 10 days data challenge. During this test no problems were encountered.

The consistency of data has been verified for several scenarios. In order to verify the proper locking, data was reconstructed after two data nodes had written data both sequentially and as well as simultaneously. Furthermore, data was successfully reconstructed during the write accesses of one or more of the data nodes. Eventually, data was reconstructed successfully after long term access. The consistency of data when a ClusterRAID device is used along with a file system was verified by writing the source code of the Linux kernel to a ClusterRAID data device and then replacing the corresponding data node with a spare one. The fact that, following this data node replacement, the kernel was successfully compiled on the replacement node indicates that data consistency is preserved during the replacement of a node, even when using a file system.

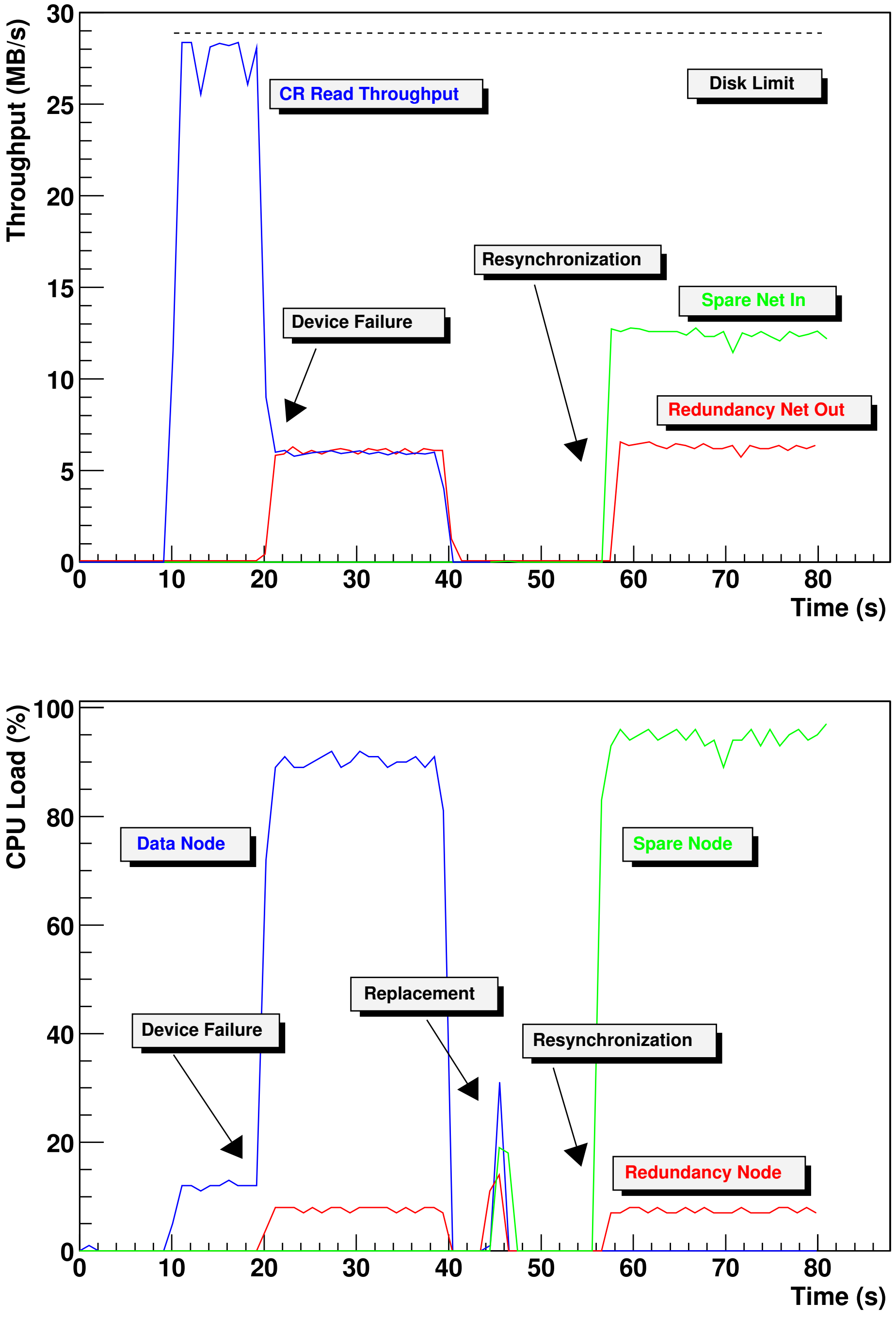

FIGURE 6.17.: System test of the ClusterRAID. See section 6.3.1 for a detailed description.

6.3.3. ClusterRAID and MFS

The ClusterRAID provides a reliable, *local* device. As the previous sections have shown, the ClusterRAID can be used along with a local file system. In order to work with the ClusterRAID in a convenient and efficient way it may be desirable to have a global view on all data in a cluster. Distributed file systems, such as PVFS or Lustre, provide such features in cluster environments. As the ClusterRAID is accessed by the standard block device interface, it can be used together with such file systems in order to have all data visible on all nodes. However, since the ClusterRAID is optimized for local access, the knowledge of *where* data resides physically, along with a corresponding scheduling policy, would be required. One option is to provide the information about the data distribution to a resource management system that can start the jobs on the nodes where they have local access to their data. Another would be to let the processes migrate to their data *automatically*. Such a service is provided, for instance, by MOSIX and its file system MFS (see section 2.2.3). For this reason, MFS has been chosen as the distributed file system for testing the ClusterRAID. It should be noted here, however, that the block device interface of the ClusterRAID also allows the use of any other distributed or parallel file system.

As MOSIX is available only for the most current kernel, the (8,2) ClusterRAID test environment was updated to kernel version 2.4.28. MOSIX was set up to use the Fast Ethernet interconnect, while the ClusterRAID used the Gigabit Ethernet connection between the nodes. This decision about which interconnect to use for the different systems was mainly influenced by the comparative ease of its respective setup. Other configurations, including the use of the same network, should also be possible.

In order to decide whether it is beneficial for the overall performance to migrate a process, MOSIX collects process statistics. The amount of time for which the measured profiling information is retained as a basis for this decision can be controlled by the so-called *decay* parameter. For the following tests this parameter was set to a level where MOSIX almost immediately reacts to remote I/O calls of the processes in question. As mentioned in section 2.2.3, the MFS provides a global view of all file systems on the MOSIX nodes. For the ClusterRAID devices on the nodes, the Third Extended File System (ext3) has been used instead of the ext2 used in previous tests. This change in file systems was made because the journalling capabilities of ext3 avoided file system checks after node crashes. The drop in disk throughput to about 14 MB/s (instead of about 27 MB/s for the 2.4.23 kernel) is due to a change in the DMA mode from Ultra-DMA (udma2) to Multiword DMA (mdma2). Details on this point are provided in Appendix A.5. However, rather than performance, the interaction of MFS and the ClusterRAID was of interest during the following tests.

In order to verify the successful interplay of MFS, MOSIX, and the ClusterRAID, a small “jumper program” has been written that accesses the data nodes in the test rig in a round robin fashion. The program always accesses its local device first, before it starts accessing remote files by means of MFS. Figure 6.18 shows the throughput of the ClusterRAID devices on the six data nodes while they are accessed by the jumper program. The blue graph shows the throughput at which the ClusterRAID devices on the data nodes deliver data to the application. Due to the sensitivity of the MOSIX kernel with the used setting, the application migrates already after accessing a few megabytes. The rate is therefore almost equal to a local access. Between accesses to the different nodes, the application is suspended for 10 seconds, hence the observed drops in the rate.

MOSIX allows binding of applications to specific nodes, in particular to their home node, i.e. the node that the application was launched on. The red graph in figure 6.18 shows a trace of the jumper

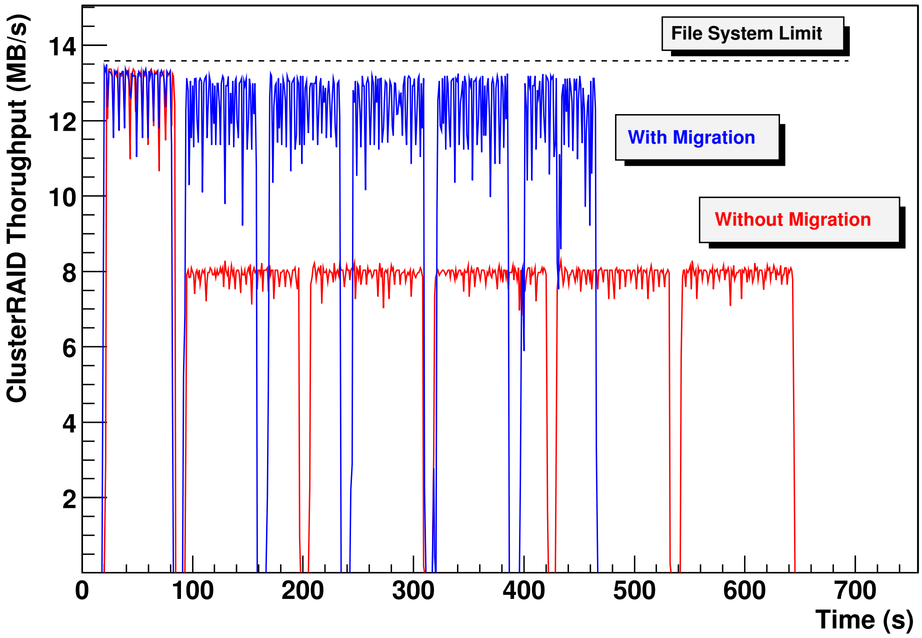

FIGURE 6.18.: MOSIX throughput improvement by process migration. See text for a detailed description.

program and the delivered ClusterRAID throughput, when the application is not allowed to be migrated to the node where the accessed data resides. Since the first data access is on a local device, there is no change in performance compared to the previous test run. However, the drop in performance due to the transport of the data over the net is obvious for the following accesses to remote nodes. This underlines once more the benefit of process migration for overall system performance.

Figure 6.19 shows a similar trace with a hard disk failure during access. In addition to the device throughput, the incoming network traffic on the application's home node is also depicted in the upper graph. The lower graph shows the CPU load during the test, in particular on the home node, the node with the faulty device (faulty node), and on one of the remaining data nodes. The other nodes are not shown for reasons of clarity. Since the first data is read from the home node's local ClusterRAID device, the network traffic is zero. The load on the node is about 10%, in good agreement with previous measurements. Once a remote device is accessed, network traffic is induced, since a part of the process always remains on its home node, e.g. in order to print out log messages. However, the part of the process responsible for I/O is migrated. The observed I/O rate is almost the same as on the application's home node and the load on the home node drops to that of an unloaded system. During the access to the files and the ClusterRAID device on the third node, the underlying device is marked faulty after around 215 seconds of operation. The ClusterRAID bandwidth drops to a few megabytes and the reconstruction imposes a heavy CPU load on the node. After about 240 seconds MOSIX migrates the jumper program back to its home node, presumably due to the high load on the reconstructing node. Hence, data is transferred over the network at the reconstruction rate. Later on, the process is migrated back to the faulty node where it finishes the decoding. The load on the

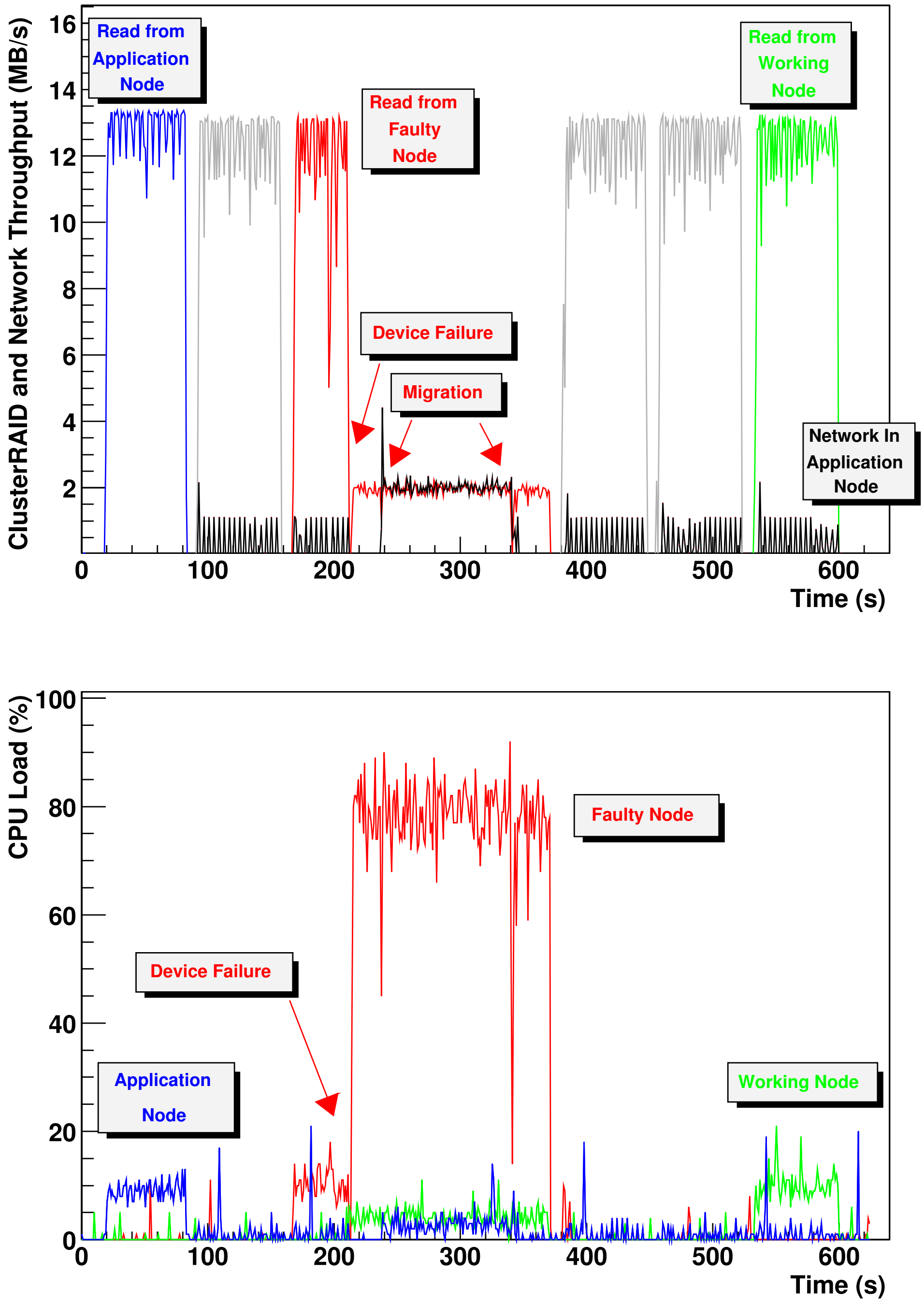

FIGURE 6.19.: System test of the ClusterRAID used with the MOSIX File System. See the text for a detailed description.

other data nodes during the reconstruction is slightly increased as well, since they must provide the data required for reconstruction. The green graphs show the throughput and the load for one of the working nodes during this test.

Aside from these tests, the consistency of data stored on a ClusterRAID device has been successfully verified by a remote Linux kernel compilation via MFS from a remote node. More MOSIX related measurements can be found in Appendix A.5.

6.4. Implemented Optimizations and Possible Improvements

Already during the development of the prototype implementation, a number of software optimizations have been applied. Some of these are the direct result of an in-depth performance analysis coupled with an identification of aspects where improvement would be beneficial [177]. This analysis and the performance results as presented in the previous sections also showed that dedicated hardware support for the most computing-intensive parts of the system may be beneficial in order to increase the performance, especially during reconstruction. Therefore, efforts have been made to port the most computing-intensive parts of the driver, i.e. the Reed-Solomon codec, to programmable hardware. The subsection on hardware found below provides only a short summary of this approach and its preliminary results. Further details on this subject can be found in [177].

6.4.1. Software

For a mass storage system, throughput is probably the most essential measure of performance. However, CPU load and network traffic should also be considered, in particular for distributed systems. The performance measurements of the prototype implementation have shown that the aggregate read performance of the ClusterRAID scales perfectly with the number of involved nodes. Write performance and the access rate in degraded mode, however, still require improvement. The reasons for the throughput limitation for these two access types are completely different. Reconstruction is obviously limited by the computing-intensive Reed-Solomon decoding. Writes on the other hand are hampered by the required read-modify-write (RMW) cycles to the underlying hard disk drives and the concurrent access to the redundancy devices. Hence, the write performance is more architecture-bound, while the performance of reconstruction is more amenable to algorithmic and processing-related improvements.

The optimization of fundamental operations, such as the multiplication over a Galois field or the XOR computation, affect almost all data paths. Moreover, such changes not only influence the throughput, but may also lower the processing overhead. In the following, a number of optimizations applied to the prototype will be presented along with a quantitative discussion of their effects.

The implementation of the ClusterRAID prototype is based on asynchronous request handling realized by callbacks. This asynchronous design applies to all request types, i.e. read, write, and reconstruct, and gives rise to a pipelining effect that speeds up the overall throughput. For instance, the throughput for reconstruction is increased by about a factor of two compared to a synchronous solution.

The analysis cited above has identified the multiplication over a Galois field as a main contributor to the CPU load during writes. If the cache of the processor is large enough to hold the required

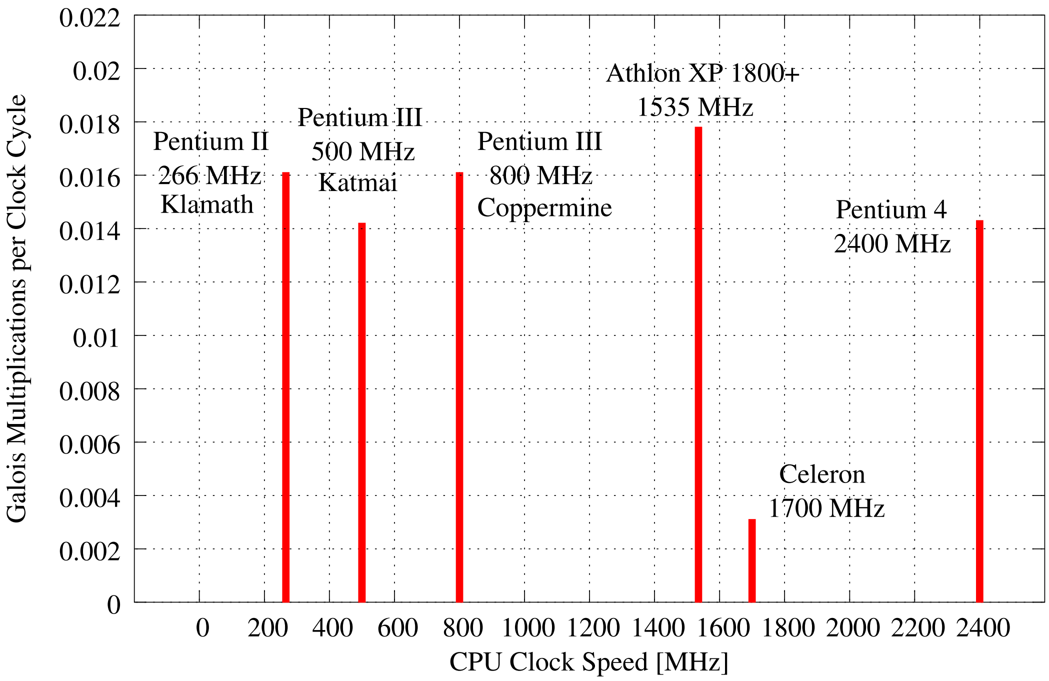

FIGURE 6.20.: Scaling behaviour of the Galois field multiplication with CPU clock speed [177]. Due to its small cache sizes, the performance of the Celeron processor is worst.

lookup-tables, each of which is 128 kilobytes in size, the Galois multiplication shows good scaling with the CPU clock speed, see figure 6.20. Hence, the rapid development of CPU clock speeds should mitigate the impact of this operation.

As the XOR'ing of symbols over whole blocks of data is frequently performed by the ClusterRAID, efforts have been made to exploit the SIMD capabilities of modern processors to lower the overhead for this operation, see also section 5.2.3. The contribution to the CPU load of the function responsible for XOR'ing data blocks could be reduced by almost a factor of 4 by this enhancement.

In order to exploit the capabilities of multiprocessor systems, the CPU-bound reconstruction of data is performed by up to four recovery threads. Using two recovery daemons instead of one improved the reconstruction throughput by 50% on the dual SMP systems used in the test rig.

The enhancement of the decoder to use the inverse of the information dispersal matrix instead of an LU decomposition reduced the computational complexity, see equation 6.1, and also improved the throughput in degraded mode, this time by a further 60%.

There are still some aspects that could be investigated to further increase the rate during reconstruction. Among these is the scattering function that distributes the requests to be reconstructed among the recovery daemons. The distribution policy has been shown to affect the rate due to the relation between the work-sleep cycles of the daemons and the number of concurrently decoding processes. This is a similar effect as discussed in section 5.3.3 concerning the degraded mode.

Given that it is architecture-bound, the improvement of write throughput is more intricate. In addition to the RAID-5 like distribution of redundancy information among all ClusterRAID nodes, as discussed in section 3.3, there are several other options to consider. In order to reduce the number of disk accesses on the redundancy side, which is a main limiting factor due to the RMW cycles, an additional request coalescing layer could be introduced. Due to the commutativity of the XOR operation, *two or more requests* for the same block could thus be merged before *one* RMW cycle corrects the data on the disk. Thus, this layer would correspond to a ClusterRAID adapted buffer cache.

Another approach would be the introduction of less strict consistency policies and write barriers, such as those used by the DRAID system or, optionally, by the DRBD. This would eliminate one of the main bottlenecks identified by figure 6.13, i.e. the need to keep requests synchronized with the underlying devices.

Reducing the overhead imposed by the network during writes or during reconstruction could be achieved by porting the underlying network block device software to network technologies and protocols more oriented to be used in clusters, such as VIA, InfiniBand, STP, or HyperSCSI to name but a few.

6.4.2. Hardware Support

Graphics Adapter

Pushed by market pressure from the video gaming industry, the evolution of modern graphics processing units (GPUs) has outpaced that of standard CPUs. While the performance of the latter closely follows Moore's law and doubles approximately every 18 months, the performance of GPUs is currently doubled every six months [178]. As of today, a modern GPU comprises about 100 million transistors, which is comparable to the numbers in modern CPUs. With increasing computing power, modern graphics adapters have also gained programmability. *Shader programs* can be loaded into the graphics adapter and are then executed by the pixel and vertex processors on the card. The architecture of GPUs, with their separate pixel pipelines, allows for an independent and explicitely parallel processing of data. This architecture maps well with the coding and decoding of the independent symbols within a data block as performed by the ClusterRAID. Since there are many applications that have a similar demand for a flexibly programmable, powerful, and cheap co-processor, a number of projects investigate ways to increase the accessibility of the GPUs on standard graphics adapters [179]. For this purpose, a hierarchy of APIs on different abstraction levels is available, ranging from high-level language extensions, such as BrookGPU [180] or NVidia's Cg [181], down to assembler. Higher level interfaces can provide a more abstract view and encapsulate programming details, but they may also reduce the available functionality. For the ClusterRAID, two approaches have been examined, namely the BrookGPU framework and the combination of OpenGL [182] and Cg. For the evaluation of a GPU-based co-processor for the Reed-Solomon coding an Albatron FX5700U graphics card equipped with an NVIDIA NV36 GPU was used. The card has 128 megabytes of memory and a nominal memory bandwidth of 14.4 GB/s. The GPU features clock rates of 300 and 475 MHz for 2D and 3D processing, respectively, and comprises four pixel pipelines for parallel data processing.

BrookGPU is an extension of standard ANSI C, but relies on a stream programming paradigm. Streams in this sense are sets of independent data that may be processed in parallel. The actual computation is performed in so-called *kernels*, which are executed on the GPU. BrookGPU is designed to exploit the parallelism of modern graphics adapters. The more complex the kernels, the less communication is required and the better the performance of BrookGPU. Although BrookGPU provides a convenient high-level programming framework, it was found to be inappropriate for the ClusterRAID for several reasons. The Galois multiplication, the basis for the Vandermonde-based Reed-Solomon codes, is more complex than a usual multiplication. However, the Galois multiplication does not require the degree of kernel-internal computation needed to amortize the high costs of setting up data transfers in and out of the graphics cards by BrookGPU. The rate at which Galois multiplications can be performed on the GPU using BrookGPU is less than 1 per microsecond and therefore around two orders of magnitude worse compared to the direct processing by the host CPU. The high setup costs, some inconvenient programming restrictions, and the poor Galois multiplication performance brought about the decision to use a lower level API, notably the Cg programming language that is also used internally by BrookGPU.

Cg stands for *C for Graphics* and is a high-level language designed to allow easy and efficient programming of modern graphics hardware. Cg supports the two most widespread APIs for accessing specialized multimedia hardware, namely OpenGL and DirectX [183]. Using these interfaces it is possible to load Cg programs into the programmable processors inside the graphics pipeline. In principle, two processors are available for external shaders: the vertex processor and the fragment processor. While the vertex processor manipulates geometrical objects described by their vertices, the fragment processor works on so-called fragments, i.e. data structures containing pixel information. For the implementation of the Reed-Solomon coder the programmable fragment processor was selected due to its more convenient programmability. The data to be encoded and the lookup tables for the logarithm and the inverse of Galois field elements are stored in textures, i.e. pixel container structures.

Preliminary measurements for the encoding step with a prototype version of a coder programmed in Cg and executed on a the graphics adapter show that the GPU outperforms the CPUs of figure 6.20 by roughly a factor of three. However, this number only holds true if the resulting data is not read back from the card, but is instead discarded. The asymmetric I/O architecture of AGP-based graphics cards renders the read-back a significant bottleneck. This drawback may be overcome with PCI Express based cards that support a symmetric data flow.

FPGA Co-processor Card

In addition to the potential usefulness of modern graphics adapters for the ClusterRAID coding, parts of the algorithm have also been implemented in a Field Programmable Gate Array (FPGA). FPGAs are programmable logic devices that comprise a number of logic blocks, I/O blocks, which provide access to the FPGA's I/O pins, and a programmable routing network for the interconnection of the blocks. For the evaluation of FPGAs within the ClusterRAID, a PCI-based evaluation board used in the ALICE High-Level Trigger Project (HLT) [184] was used. This board is equipped with an Altera [185] APEX EP20K400EFC672-1x FPGA comprising 16,640 logic blocks and 212,929 on-chip memory bits. For the data transfers into and out of the card by means of the PCI bus, slightly modified versions of the existing DMA prototype designs for the HLT were needed.

As both encoding and decoding can be performed by a number a multiplications, the basic building block of the FPGA implementation was the design of an encoder. This block is able to perform a multiplication of an 8-bit data word with a constant number. The constant is a specific element of the information dispersal matrix and is identified by a node's unique ClusterRAID ID and the redundancy device, for which data is to be encoded. This approach avoids the use of one lookup table otherwise required to perform the multiplication. The constant can be set once during the initialization of the FPGA. As the PCI bus used in this setup transfers 32-bit data words, four coder blocks can be used in parallel. In addition to this 4-way coder module, an incoming and outgoing FIFO as well as a small arbiter complete the prototype design.

The PCI Shared Memory Interface library and driver (PSI) [176, 186] were used to setup the card and initiate data transfers. The maximum achieved throughput for encoding was around 15 MB/s. It should be noted, however, that this is a preliminary result and that due to time constraints no efforts have been made yet to optimize the design. The bottleneck in the data path was not the coder but the DMA functionality, which was neither intended nor optimized for use with the ClusterRAID. However, it could be shown that it is feasible to port the coding algorithm and to realize the whole data path using FPGAs. Aside from a customized data transport design, the next steps would include the actual implementation of the decoder, a proposal of which has already been developed [177], and the integration of the hardware components with the software driver.

7. Conclusion and Outlook

> *There are two kinds of people, those*
> *who finish what they start and so on.*
> Robert Byrne

This thesis has presented the architecture and a prototype implementation of the ClusterRAID, a novel distributed mass storage architecture for clusters. The basic concept of this system is to convert the local hard disk drive into a reliable device while preserving its block interface.

The ClusterRAID implements an additional layer in the I/O data path between user applications and locally attached mass storage devices. From the user data point of view, the ClusterRAID realizes a shared-nothing architecture: during normal operation every node in the cluster uses its local hard disk drive independently and asynchronously from all other nodes. Successful read requests are served from the underlying hard drive. Therefore, the read bandwidth compares with the nominal disk speed, and due to the local access no network transactions are required, if the underlying device is operational. Like read requests, write requests are also forwarded to the constituent device. However, they additionally trigger the generation of redundancy information, which can be used to reconstruct the user data in case of device failures. By storing the redundant information remotely on other nodes, the ClusterRAID protects against data loss by complete node failures and improves data availability, as the distribution of redundancy information allows for the reconstruction of data to an arbitrary node.

The approach of keeping the user data of individual nodes separate and of optimizing the read access is adapted to the characteristics of a broad range of parallelizable applications. Examples include all kinds of data analyses, data mining, or information retrieval systems. For these applications, that are characterized by a bias towards read accesses, the ClusterRAID can provide the maximum performance of the underlying hardware.

Aside from the generation of redundancy information in the case of write accesses, the interception and supervision of all requests to the constituent device exhibits an entry point for fault-tolerance mechanisms. The ability to deal with, possibly multiple, component failures is a fundamental requirement for all cluster-based mass storage systems and is therefore also provided by the ClusterRAID architecture. The reconstruction of data is initiated when the underlying device reports a failing request. In this case, the ClusterRAID layer decodes the requested data online and transparently to the application, using the redundancy information and the remote data associated with this particular redundancy ensemble.

A prototype of the ClusterRAID architecture has been implemented as a set of kernel modules for the GNU/Linux operating system. The design is modular in that the request handling and the coding functionality are realized by different entities. For the prototype, a codec module using an adapted version of Reed-Solomon codes has been implemented. This codec allows the configuration of the number of tolerable device failures and requires only the theoretical minimum of space overhead.

In a number of performance and functional benchmarks, such as the simulation of error scenarios or the cooperation of the ClusterRAID with local and distributed file systems, the validity and the feasibility of the concept have been shown.
The measurements, however, also indicated that further optimization is possible to reduce the computing overhead, in particular during reconstruction, and to increase the throughput, especially for write requests. As a first step, a number of software optimizations have been applied, such as the totally asynchronous reconstruction mechanism based on kernel threads or the usage of special processor features. Besides the port of the driver to the current 2.6 series of the Linux kernel with its completely new I/O subsystem, the uniform distribution of redundancy information on all ClusterRAID nodes, the implementation of request coalescing, and the evaluation of alternative coding algorithms would complement these software-based improvements. In addition to software optimizations, the preliminary evaluation of hardware support for the coding has shown that it may be worthwhile to pursue hardware-based improvements to the system. As all write requests in the ClusterRAID require almost subsequent read and write accesses to the same disk block, an intelligent disk controller accounting for this special access behaviour will also improve the write performance.
Although the ClusterRAID prototype already contains all necessary building blocks, a meta instance, that supervises the global state of the ClusterRAID, would provide a good complement for the productive use of the system. Among the tasks of such an instance could be the automatic replacement of faulty nodes with spare ones and the initiation of data reconstruction on the newly inserted node. Also included could be a solution to the issue of faulty nodes which still hold a lock and may block other nodes. For this, the meta instance could incorporate already existing distributed lock managers. Finally, the instance could manage data consistency during system crashes by realizing atomic block writes or by acting as the coordinator in two-phase commit protocols.

A. Appendix

A.1. The Galois Field $GF(2^4)$

code word	polynomial in x mod h(x)	power of β
0000	0	-
1000	1	β^0
0100	x	β^1
0010	x^2	β^2
0001	x^3	β^3
1100	$x^4 = 1+x$	β^4
0110	$x^5 = x+x^2$	β^5
0011	$x^6 = x^2+x^3$	β^6
1101	$x^7 = 1+x+x^3$	β^7
1010	$x^8 = 1+x^2$	β^8
0101	$x^9 = x+x^3$	β^9
1110	$x^{10} = 1+x+x^2$	β^{10}
0111	$x^{11} = x+x^2+x^3$	β^{11}
1111	$x^{12} = 1+x+x^2+x^3$	β^{12}
1011	$x^{13} = 1+x^2+x^3$	β^{13}
1001	$x^{14} = 1+x^3$	β^{14}

TABLE A.1.: The elements of $GF(2^4)$ constructed using $h(x) = 1+x+x^4$.

A.2. A Production Quality Distributed RAID

In order to prove the feasibility of the Distributed RAID concept set out in section 2.5, a production quality system was built. This system now serves as the central file server for the Kirchhoff Institute of Physics in Heidelberg [163]. The system comprises three backend nodes with six 200 gigabytes SATA hard drives (in this case Seagate ST3200822AS) each. Five of the six hard drives on each node are the constituent devices of a local hardware RAID-5. The RAID controller used is a Highpoint RocketRAID 1820 [187]. The remaining sixth disk is used as a hot spare. All nodes, including the frontend node, are based on an Intel D865GBF motherboard with a Pentium 4 CPU running at 2.4 GHz. Each node has 1 gigabyte main memory and an onboard Gigabit Ethernet interface (Intel EEPRO1000). The nodes are running a Debian-based Linux [188] with kernel version 2.4.26. The system is interconnected by a Cisco 4500 GBit switch [189]. A photograph of the assembled system

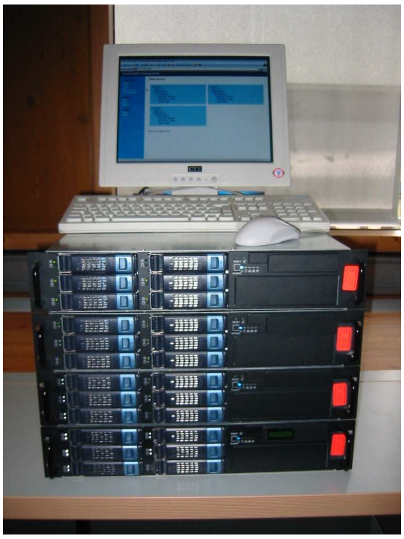

FIGURE A.1: Photograph of the production quality system of a Distributed RAID. The lower three nodes (backends) have local hardware RAIDs, each of which comprises six 200 GB SATA disk drives (five data disks and one spare device). The upper node is used as the frontend, which imports the remote devices via ENBD. In this specific configuration, the total raw capacity of the system is 3.6 terabytes.

before commissioning is shown in figure A.1.
The backend nodes export their local RAID devices via ENBD to the frontend. Each of these devices has a net capacity of around 800 gigabytes. On the frontend server, the imported ENBD devices themselves form a local software RAID-5. Due to the small number of imported devices, the space overhead in this configuration is rather large: of the total gross capacity of about 3.6 terabytes (3 nodes times six disks with 200 gigabytes each) only 1.6 terabytes can be used to actually store user data. This ratio of usable to raw capacity of 44% can be improved to 55% if the hot spare disk becomes part of the hardware RAIDs on the backend servers or to 66% if five backend servers instead of three are used.
By using local RAIDs on the backend servers the whole system can tolerate the loss of one disk on each backend node and, in addition, the loss of a complete backend node. Faulty devices can be exchanged at runtime due to the hotplug capability of the SATA devices. Performance numbers of the system are summarized in table A.2.

READ	File Size	BE Throughput	BE CPU Load	FE Throughput	FE CPU Load
	2,048 MB	117.4 MB/s	31%	88.9 MB/s	32%
	4,096 MB	111.3 MB/s	31%	79.6 MB/s	35%
	8,192 MB	111.8 MB/s	32%	79.6 MB/s	36%

WRITE	File Size	BE Throughput	BE CPU Load	FE Throughput	FE CPU Load
	2,048 MB	58.3 MB/s	37%	36.8 MB/s	32%
	4,096 MB	51.3 MB/s	35%	30.9 MB/s	25%
	8,192 MB	48.5 MB/s	34%	28.3 MB/s	22%

TABLE A.2.: Performance numbers of the production quality Distributed RAID as measured with the bonnie++ I/O benchmark. BE denotes the backend servers and the throughput numbers are for the local hardware RAID. FE denotes the frontend servers and the throughput numbers are for the RAID over ENBD devices.

For the configuration of the Distributed RAID an XML-based configuration tool with a web interface has been developed [190]. A more detailed description of the system can also be found in [191].

A.3. The DWARW Module

As a by-product of the development of the ClusterRAID driver, a block device monitor module has been implemented, called the Device Write And Read Access Watcher (DWARW). Similar to the ClusterRAID module this is a proxy device that remaps all requests to an underlying block device. During this remapping, however, the DWARW extracts information about the individual requests from the corresponding `buffer_head`.

Figure A.2 shows a schematic overview of the monitoring system which comprises the `dwarw` Linux kernel module, the `elv` data extraction module, a small configuration program, called `dwarw-ctl`, and the user space program `palantir` that fetches the monitoring data from kernel space. The `dwarw` kernel module provides the block device interface and forwards the requests to an assigned underlying block device. The driver implements a virtual device, i.e. it controls no real hardware, and as such maintains no request queue, but registers a `make_request_fn` function instead. In this function the desired request information is extracted and the requests are remapped and reissued to the target device. In contrast to the ClusterRAID driver, the DWARW module is not required to provide callbacks to check the completion status of individual requests.

The registration of data extraction modules uses a similar interface as that of the registration of codec modules in the ClusterRAID. This modularity allows for an easy exchange and evaluation of different approaches to the collection and transport of monitoring data to user space, even during runtime. It is possible to use multiple extraction modules, or none, registered at the same time. The `elv` module is the only data extraction module currently implemented. This module stores the monitoring data in two separate lists. While one of these lists is used to add newly gathered information, the other list is readout by a user space program at regular intervals. Every time the read out list is empty, the lists change their roles. The actual read out is triggered by ioctls and the data transfer is realized by a kernel to user space copy step. If it turns out that this type of data transfer is too costly, it may be worth examining alternative approaches such as memory areas mapped in both user and kernel space. The `palantir` user space program simply writes the gathered data to the standard output.

As an alternative to the output modules, information can also be extracted directly by the `dwarw` module. In the current implementation, however, the module only counts the read and write requests per device and provides the statistics under `/proc/dwarw`.

Due to its proxy functionality the DWARW module can also be used to provide multiple interfaces to a single hardware device. This allows, for instance, the measurement of the behaviour of multiple processes accessing the same hard drive, but using their private buffer cache, since the access is performed by means of different special device files and the kernel considers them to be separate devices. This feature has been used for the benchmarks in chapter 6.

A.4. The NetIO Network Benchmark

Instead of using existing network benchmarks, the small tool `netIO` has been developed which fulfills some of the requirements not met by other programs. In addition to the standard protocols TCP and UDP, `netIO` also supports measurements with the Scheduled Transfer Protocol (STP). NetIO

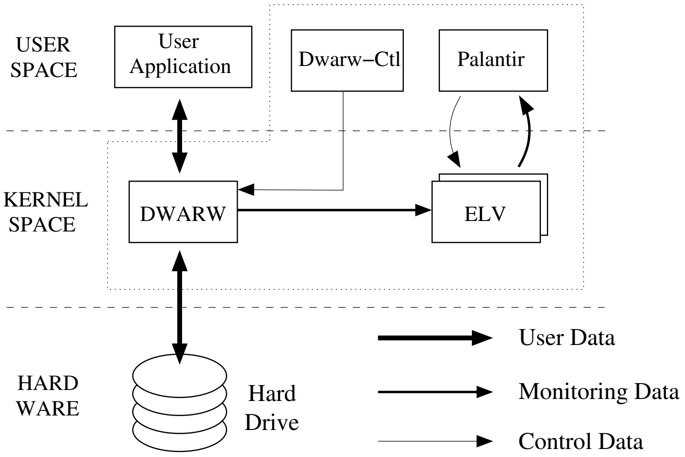

FIGURE A.2.: Schematic view of the DWARW monitoring system. The four basic components are encircled by the dotted line.

can use the `/proc` output of kernels which have the Precise Accounting Patch applied. Moreover, `netIO` has support for data verification, i.e. the program can check if the data received is identical to the data that has been sent. The available options are listed in table A.3.

A.5. MOSIX related Measurements

For the measurements in section 6.3.3, the MOSIX decay parameters were adapted to consider only short term process statistics for the migration decision. Hence, processes that accessed files on remote hosts were migrated almost instantly to the nodes on which the accessed data resided. Figure A.3 shows the throughput of the ClusterRAID devices on the MOSIX nodes if the decay parameters are left at their default values. As in section 6.3.3, the jumper program accessed all six data nodes in a round-robin fashion via MFS. The blue graphs denote the throughput of the local ClusterRAID devices, while the red curve is the incoming network traffic on the application's home node.

Since the first data to be accessed resided on the home node, the ClusterRAID throughput is at its maximum and the network traffic is zero. Once the access is completed, and after a ten seconds pause, the program starts accessing data on a remote node. Due to the default parameters, however, the process is not migrated immediately, but the data is transferred over the network instead. Hence, the bandwidth available to the application is limited by the network transmission in this case. Once the application migrates to the node which holds the data, the device throughput increases due to the local access and the network rate drops to below 1 MB/s. The reason for the remaining network traffic is the communication of the migrated process with a stub process left at the home node. This stub process is used to perform all actions that cannot be migrated, such as console printouts.

If the process starts accessing data on a third node, the procedure is repeated. The only difference to the very first migration is the fact that these later migrations do not originate from the application's home node. Therefore, the incoming network traffic on the home node stays at its low level. The increased density of the network peaks after a process is migrated is due to the status output of the jumper program, since its frequency is related to the transfer rate. For reasons of completeness and

`[-s|-c]`	Instance is client (sender) or server (receiver). Mandatory.
`-p [TCP|UDP|STP]`	Protocol selection. Mandatory.
`-b size`	The block size to use in the test.
`-ss size`	The socket buffer size.
`-d`	If used, the `TCP_NODELAY` option is set to 0, i.e. delay is activated.
`-pp port`	The port number to use.
`-ip ip-address`	The IP address to connect to. Mandatory on client side.
`-i number`	The number of iterations the test shall be repeated.
`-t time`	The timeout to be used in UDP transmissions.
`-v`	Check for transmissions errors.
`-vv`	Verbose mode.
`-cc`	Measures the CPU load. Precise Accounting Patch required.
`-r`	Prints out the sending rate.
`-ll`	Measures the transfer latency (one way).

TABLE A.3.: Available options of the netIO benchmark.

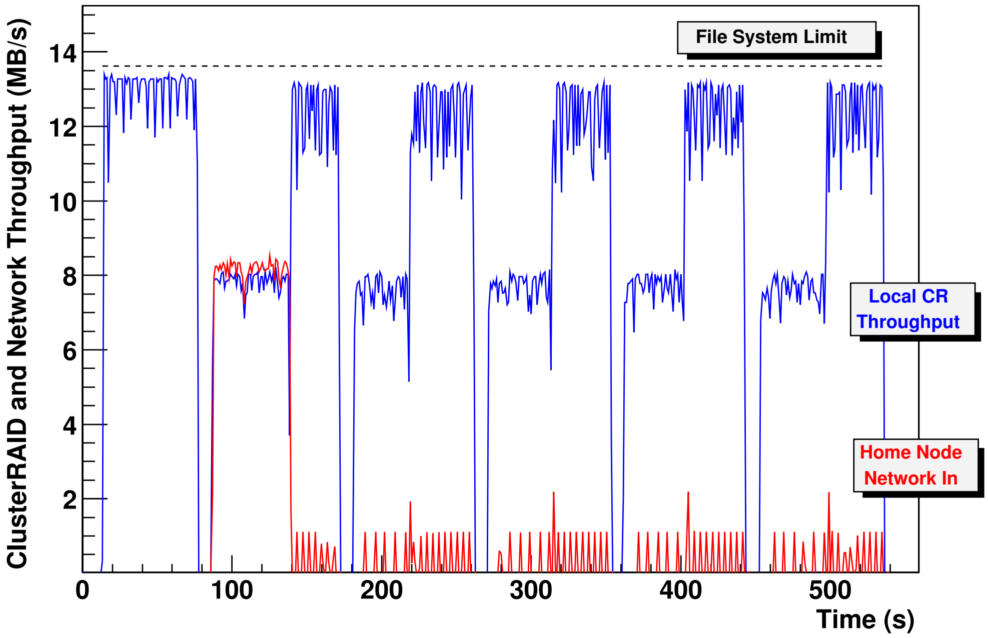

FIGURE A.3.: MOSIX process migration due to I/O load on an (8,2) ClusterRAID system.

MOSIX I/O Performance				
Kernel	File System	CR (8,2)	Read	Write
2.4.23	none	no	28.8 MB/s	27.9 MB/s
2.4.23	none	yes	27.8 MB/s	3.7 MB/s
2.4.23	ext2	no	28.3 MB/s	28.7 MB/s
2.4.23	ext2	yes	28.2 MB/s	3.4 MB/s
2.4.28	ext2	no	14.8 MB/s	14.7 MB/s
2.4.28 (MOSIX)	none	no	14.3 MB/s	10.6 MB/s
2.4.28 (MOSIX)	ext2	no	13.8 MB/s	14.0 MB/s
2.4.28 (MOSIX)	none	yes	13.8 MB/s	2.9 MB/s
2.4.28 (MOSIX)	ext3	no	14.0 MB/s	13.6 MB/s
2.4.28 (MOSIX)	ext3	yes	13.6 MB/s	2.4 MB/s

TABLE A.4.: MOSIX I/O Performance.

comparison, table A.4 summarizes the I/O transfer rates of the test bed for the various combinations of kernel versions and file systems, used with and without the MOSIX extension, and accessed directly or by means of the ClusterRAID. The low bandwidth in the tests with the 2.4.28 kernel is caused by problems with activating the Ultra-DMA mode of the hard disk under this particular kernel.

Bibliography

[1] The Top500 Supercomputer Sites.
URL `http://www.top500.org`

[2] T. Anderson, D. Culler, and D. A. Patterson, *A Case for NOW (Network of Workstations)*, in IEEE Micro, 15(1):54–64, February 1995.

[3] V. Lindenstruth and A. Wiebalck, *Verfahren und Vorrichtung zum Sichern von Daten auf mehreren unabhängigen Schreib-Lese-Speichern*, filed with the German Patent and Trademark Office, serial no. DE 103 50 590.3-53, October 2003.

[4] V. Lindenstruth and A. Wiebalck, *Method and Apparatus for Enabling High-Reliability Storage of Distributed Data on a Plurality of Independent Storage Devices*, filed with the U.S. Patent and Trademark Office, serial no. US 10/977.725, October 2004.

[5] V. Lindenstruth and A. Wiebalck, *Method and Apparatus for Enabling High-Reliability Storage of Distributed Data on a Plurality of Independent Storage Devices*, filed with the European Patent Office, serial no. PCT/EP 2004/012314, October 2004.

[6] G. Pfister, *In Search of Clusters*, Prentice Hall, 2 edn., 1998.

[7] M. Baker and R. Buyya, *Cluster Computing: The Commodity Supercomputer*, in Software - Practice & Experience, 29(6):551–576, May 1999.

[8] G. E. Moore, *Cramming More Components onto Integrated Circuits*, in Electronics, 38(8):114–117, April 1965.

[9] R. J. T. Morris and B. J. Truskowski, *The Evolution of Storage Systems*, in IBM Systems Journal, 42(2):205–217, 2003.

[10] M. N. Baibich, J. M. Broto, A. Fert, F. N. V. Dau, F. Petroff, P. Eitenne, G. Creuzet, A. Friederich, and J. Chazelas, *Giant Magnetoresistance of (001) Fe/(001) Cr Magnetic Superlattices*, in Physical Review Letters, 61(21):2472–2475, November 1988.

[11] G. Binasch, P. Grünberg, F. Saurenbach, and W. Zinn, *Enhanced Magnetoresistance in Layered Magnetic Structures with Antiferromagnetic Interlayer Exchange*, in Physical Review B, 39(7):4828–4830, March 1989.

[12] E. Grochowski and R. D. Halem, *Technological Impact of Hard Disk Drives on Storage Systems*, in IBM Systems Journal, 42(2):338–346, 2003.

[13] V. Skumryev, S. Stoyanov, Y. Zhang, G. Hadjipanayis, D. Givord, and J. Nogues, *Beating the Superparamagnetic Limit with Exchange Bias*, in Nature, 423(6942):850–853, June 2003.

[14] D. A. Thompson and J. S. Best, *The Future of Magnetic Data Storage Technology*, in IBM Journal of Research and Development, 44(3):311–322, May 2000.

[15] M. H. Kryder, *Future Trends in Magnetic Storage Technology*, in *Digest of Technical Papers 2nd North American Perpendicular Magnetic Recording Conference and the 6th Perpendicular Magnetic Recording Conference (NAPMRC 2003)*, 68, January 2003.

[16] C. A. Ross, *Patterned Magnetic Recording Media*, in Annual Review of Materials Research, 31:203–235, August 2001.

[17] J. Menon, *Grand Challenges facing Storage Systems*, in *Computing in High Energy and Nuclear Physics Conference 2004 (CHEP 2004)*, Interlaken, Switzerland, September 2004.

[18] A. Acharya, M. Uysal, and J. Saltz, *Active Disks: Programming Model, Algorithms and Evaluation*, in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1998)*, 81–91, October 1998.

[19] E. Riedel, *Active Disks - Remote Execution for Network-Attached Storage*, Ph.D. thesis, Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, November 1999.

[20] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, *Active Disks for Large-Scale Data Processing*, in IEEE Computer, 34(6):68–74, June 2001.

[21] X. Ma and A. L. N. Reddy, *MVSS: An Active Storage Architecture*, in IEEE Transactions on Parallel and Distributed Systems, 14(10):993–1005, October 2003.

[22] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufmann, 2 edn., 1998.

[23] S. Tommesani, *The MMX Primer*.
URL `http://www.tommesani.com/MMXPrimer.html`

[24] S. Tommesani, *Introduction to SIMD Programming*.
URL `http://www.tommesani.com/Docs.html`

[25] B. Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*.
URL `http://www.x86.org/articles/sse_pt1/simd1.htm`

[26] C. Zdebel and S. Solotko, *The AMD64 Computing Platform: Your Link to the Future of Computing (AMD64 Whitepaper)*, May 2003.
URL `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/30172C.pdf`

[27] Intel Extended Memory 64 Technology.
URL `http://www.intel.com/technology/64bitextensions`

[28] Intel Website.
URL `http://www.intel.com`

[29] The PCI Special Interest Group (PCI-SIG), *PCI Local Bus Specification*.
URL `http://www.pcisig.com/specifications/conventional`

[30] The PCI Special Interest Group (PCI-SIG), *PCI-X Local Bus Specification*.
URL `http://www.pcisig.com/specifications/pcix_20`

[31] The PCI Special Interest Group (PCI-SIG), *PCI Express Bus Specification*.
URL `http://www.pcisig.com/specifications/pciexpress`

[32] Myricom and Myrinet Homepage.
URL `http://www.myri.com`

[33] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, *The Quadrics Network: High Performance Clustering Technology*, in IEEE Micro, 22(1):46–57, January-February 2002.

[34] IEEE Standard for Scalable Coherent Interface (SCI), IEEE Standard 1596-1992, October 1992.

[35] InfiniBand Trade Association, *InfiniBand Architecture Specification*.
URL `http://www.infinibandta.org`

[36] R. M. Metcalfe and D. R. Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, in Communications of the ACM, 19(5):395–404, July 1976.

[37] Information Sciences Institute, University of Southern California, *IETF RFC 793: Transmission Control Protocol*, September 1981.
URL `http://www.ietf.org/rfc/rfc793.txt`

[38] G. Ciaccio, *Optimal Communication Performance on Fast Ethernet with GAMMA*, in *Parallel and Distributed Processing*, vol. 1388 of *Lecture Notes in Computer Science*, 534–548, Springer, 1998.

[39] American National Standard for Information Technology: ANSI NCITS 337-2000, Scheduled Transfer Protocol (ST), April 2000.

[40] The Free Software Foundation (FSF), *The GNU Operating System*.
URL `http://www.gnu.org`

[41] The Linux Kernel Archives.
URL `http://www.kernel.org`

[42] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo, *Overview of Massively Parallel Operating System Kernel SCore*, Tech. Rep. TR-93003, Real World Computing Partnership, Tsukuba, Japan, 1993.

[43] The Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*.
URL `http://www.mpi-forum.org`

[44] MPICH Homepage.
URL `http://www-unix.mcs.anl.gov/mpi/mpich`

[45] LAM/MPI Homepage.
URL `http://www.lam-mpi.org`

[46] V. S. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, in Concurrency: Practice and Experience, 2(4):315–339, December 1990.

[47] LHC, *The Large Hadron Collider*.
URL `http://web.cern.ch/lhc`

[48] CERN, *European Laboratory for Particle Physics*.
URL `http://www.cern.ch`

[49] The MONARC Architecture Group, *Regional Centers for LHC Computing*.
URL `http://monarc.web.cern.ch`

[50] ATLAS: A Toroidal LHC ApparatuS.
URL `http://atlas.web.cern.ch/Atlas/`

[51] S. Bethke, M. Calvetti, H. F. Hoffmann, D. Jacobs, M. Kasemann, and D. Linglin, *Report of the Steering Group of the LHC Computing Review*, Tech. Rep. CERN/LHCC/2001-004 CERN/RRB-D 2001-3, European Organization for Nuclear Research (CERN), 2001.

[52] A. Reinefeld and V. Lindenstruth, *How to build a High-Performance Compute Cluster*, in *Proceedings of the 30th International Workshops on Parallel Processing (ICPP 2001 Workshops)*, 221–230, Valencia, Spain, September 2001.

[53] The Google Web Search Engine.
URL `http://www.google.com`

[54] L. A. Barroso, J. Dean, and U. Hölzle, *Web Search for a Planet: The Google Cluster Architecture*, in IEEE Micro, 23(2):22–28, March-April 2003.

[55] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, *OceanStore: An Architecture for Global-scale Persistent Storage*, in *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 190–201, November 2000.

[56] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, *Wide-area Cooperative Storage with CFS*, in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, 202–215, Chateau Lake Louise, Banff, Canada, October 2001.

[57] P. Druschel and A. Rowstron, *PAST: A Large-scale, Persistent Peer-to-Peer Storage Utility*, in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 75–80, Elmau, Germany, May 2001.

[58] G. A. Gibson and R. van Meter, *Network Attached Storage Architecture*, in Communications of the ACM, 43(11):37–45, November 2000.

[59] A. Benner, *Fibre Channel: Gigabit Communications and I/O for Computer Networks*, McGraw Hill, 1996.

[60] Storage Networking Industry Association (SNIA), *iSCSI Technical White Paper*.
URL `http://www.snia.org/tech_activities/ip_storage/iscsi`

[61] Storage Networking Industry Association (SNIA), *Internet Fibre Channel Protocol (iFCP) - A Technical Overview*.
URL `http://www.snia.org/tech_activities/ip_storage/ifcp`

[62] Storage Networking Industry Association (SNIA), *The Emerging FCIP Standard for Storage Area Network Connectivity Across TCP/IP Networks*.
URL `http://www.snia.org/tech_activities/ip_storage/fcip`

[63] Data Storage Institute Singapore, *HyperSCSI Protocol Specifications*.
URL `http://www.dsi.a-star.edu.sg/research/hyper_document.html`

[64] The Internet Engineering Task Force.
URL `http://www.ietf.org`

[65] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, *IETF RFC 3720: Internet Small Computer Systems Interface (iSCSI)*, April 2004.
URL `http://www.ietf.org/rfc/rfc3720.txt`

[66] D. A. Patterson, G. A. Gibson, and R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 109–116, Chicago, Illinois, USA, June 1988.

[67] J. M. May, *Parallel I/O for High Performance Computing*, Academic Press, 2001.

[68] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, *Design and Implementation of the Sun Network Filesystem*, in *Proceedings of Summer 1985 USENIX Conference*, 119–130, Portland, Oregon, USA, June 1985.

[69] Sun Microsystems, Inc., *IETF RFC 1410: External Data Representation Standard (XDR)*, June 1987.
URL `http://www.ietf.org/rfc/rfc1410.txt`

[70] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle, *The Direct Access File System*, in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, April 2003.

[71] D. Cameron and G. Regnier, *Virtual Interface Architecture*, Intel Press, 2002.

[72] A. Barak, S. Guday, and R. Wheeler, *The MOSIX Distributed Operating System, Load Balancing for UNIX*, vol. 672 of *Lecture Notes in Computer Science*, Springer, 1993.

[73] L. Amar, A. Barak, A. Eizenberg, and A. Shiloh, *The MOSIX Scalable Cluster File System for Linux*, July 2000.
URL `http://www.mosix.org`

[74] L. Amar, A. Barak, and A. Shiloh, *The MOSIX Parallel I/O System for Scalable I/O Performance*, in *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 495–500, Cambridge, Massachusetts, USA, November 2002.

[75] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, *PVFS: A Parallel File System For Linux Clusters*, in *Proceedings of the 4th Annual Linux Showcase and Conference*, 317–327, Atlanta, Georgia, USA, October 2000.

[76] IEEE and Open Group Standard for Information Technology – Portable Operating System Interface (POSIX), IEEE Standard 1003.1, 2004 Edition, May 2004.

[77] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, 1999.

[78] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. Swanson, *Improved Read Performance in a Cost-Effective, Fault-Tolerant Parallel Virtual File System (CEFT-PVFS)*, in *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 730–735, Tokyo, Japan, May 2003.

[79] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. Swanson, *Design, Implementation and Performance Evaluation of A Cost-Effective, Fault-Tolerant Parallel Virtual File System*, in *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2003)*, New Orleans, Louisiana, USA, September 2003.

[80] Cluster File Systems Inc., *Lustre: A Scalable, High-Performance File System*, White Paper, Version 1.0, November 2002.
URL `http://www.lustre.org/docs/whitepaper.pdf`

[81] F. Schmuck and R. Haskin, *GPFS: A Shared-Disk File System for Large Computing Clusters*, in *Proceedings of the First Conference on File and Storage Technologies (FAST 2002)*, 231–244, Monterey, California, USA, January 2002.
URL `citeseer.ist.psu.edu/schmuck02gpfs.html`

[82] IBM Corporation, *GPFS Primer for Linux Clusters*, March 2003.
URL `http://www-1.ibm.com/servers/eserver/clusters/whitepapers/-gpfs_linux_primer.html`

[83] IBM Corporation, *An Introduction to GPFS v1.3 for Linux, White Paper*, June 2003.
URL `http://www-1.ibm.com/servers/eserver/clusters/whitepapers/-gpfs_linux_intro.html`

[84] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*, in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, 29–43, Bolton Landing, New York, USA, October 2003.

[85] R. W. Watson and R. A. Coyne, *The parallel I/O architecture of the High Performance Storage System (HPSS)*, in *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems*, 27–44, September 1995.

[86] The Cern Advanced STORage Manager (CASTOR).
URL `http://castor.web.cern.ch/castor`

[87] J.-P. Baud, B. Coutourier, C. Curran, J.-D. Durand, E. Knezo, S. Occhetti, and O. Barring, *CASTOR Status and Evolution*, in *Proceedings of the Computing in High Energy and Nuclear Physics Conference (CHEP 2003)*, La Jolla, California, March 2003.

[88] IBM Corporation, *Storage Tank – A Distributed Storage System*, January 2002.
URL `www.almaden.ibm.com/StorageSystems/file_systems/storage-_tank/ExtStorageTankPaper01_24_02.pdf`

[89] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1994.

[90] M. Stonebraker, *Operating System Support for Database Management*, in Communications of the ACM, 24(7):412–418, July 1981.

[91] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, *Scale and Performance in a Distributed File System*, in ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[92] P. Machek, *The Network Block Device*.
URL `http://nbd.sourceforge.net`

[93] P. T. Breuer, A. M. Lopez, and A. G. Ares, *The Enhanced Network Block Device*, in Linux Journal, May 2000.

[94] T. Dierks and C. Allen, *IETF RFC 2246: The Transport Layer Security Protocol (TLS)*, January 1999.
URL `http://www.ietf.org/rfc/rfc2246.txt`

[95] P. Reisner, *DRBD: Festplattenspiegelung übers Netzwerk für die Realisierung hochverfügbarer Server unter Linux*, Master's thesis, Faculty of Science and Informatics of the Vienna University of Technology, Vienna, May 2000.

[96] K. Hwang, H. Jin, and R. S. C. Ho, *Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing*, in IEEE Transactions on Parallel and Distributed Systems, 13(1):26–44, January 2002.

[97] G. F. Pfister, *The Varieties of Single System Image*, in *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, 59–63, Princeton, New Jersey, USA, October 1993.

[98] K. Hwang, H. Jin, E. Chow, C. Wang, and Z. Xu, *Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space*, in IEEE Concurrency, 7(1):60–69, January-March 1999.

[99] E. K. Lee and C. A. Thekkath, *Petal: Distributed Virtual Disks*, in *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996)*, 84–92, Cambridge, Massachusetts, USA, October 1996.

[100] H. I. Hsiao and D. DeWitt, *Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines*, in *Proceedings of the 6th International Data Engineering Conference*, 456–465, 1990.

[101] C. A. Thekkath, T. Mann, and E. K. Lee, *Frangipani: A Scalable Distributed File System*, in *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP 1997)*, 224–237, Saint-Malo, France, October 1997.

[102] N. Talagala, S. Asami, D. A. Patterson, and K. Lutz, *Tertiary Disk: Large Scale Distributed Storage*, Tech. Rep. UCB-CSD-98-989, University of California, Berkeley, USA, 1998.

[103] R. Y. Wang and T. E. Anderson, *xFS: A Wide Area Mass Storage File System*, in *Proceedings of the 4th Workshop on Operating Systems*, 71–78, Napa, California, USA, October 1993.

[104] A. D. Marco, G. Chiola, and G. Ciaccio, *Using a Gigabit Ethernet Cluster as a Distributed Disk Array with Multiple Fault Tolerance*, in *Proceedings of the 28th Conference on Local Networks (LCN 2003), Workshop on High-Speed Local Networks (HSLN)*, 605–613, Bonn, Germany, October 2003.

[105] 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 (CCGrid 2003)*, 266–275, Tokyo, Japan, May 2003.

[106] The Heartbeat Project.
URL `http://www.linux-ha.org/heartbeat`

[107] The Linux High Availability Project.
URL `http://www.linux-ha.org`

[108] P. T. Breuer and A. Wiebalck, *Intelligent Networked Software RAID*, in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2005) as part of the 23rd IASTED International Multi-Conference on Applied Informatics*, 517–522, Innsbruck, Austria, February 2005.

[109] R. Coker, *The bonnie++ I/O benchmark*.
URL `http://www.coker.com.au/bonnie++`

[110] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, *RAID: High-Performance, Reliable Secondary Storage*, in ACM Computing Surveys, 26(2):145–185, June 1994.

[111] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, and W. Litwin, *Reliability Mechanisms for Very Large Storage Systems*, in *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS 2003, MSST 2003)*, 146–156, San Diego, California, USA, April 2003.

[112] H. Weatherspoon and J. D. Kubiatowicz, *Erasure Coding vs. Replication: A Quantitative Approach*, in *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 328–337, Cambridge, Massachusetts, USA, March 2002.

[113] OMG Unified Modeling Language Specification, Version 1.5, March 2003.
URL `http://www.uml.org`

[114] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, 2 edn., January 2001.

[115] J. H. Hartman, I. Murdock, and T. Spalink, *The Swarm Scalable Storage System*, in *19th IEEE International Conference on Distributed Computing Systems*, 74–81, Austin, Texas, USA, June 1999.

[116] Panasas Inc., *The Panasas ActiveScale File System*.
URL `http://www.panasas.com/panfs.html`

[117] ATA SMART Feature Set Commands.
URL `http://www.t13.org`

[118] SCSI "Mode Sense" Code.
URL `http://www.t10.org`

[119] W. Weibull, *A Statistical Distribution Function of Wide Applicability*, in Journal of Applied Mechanics, 18:293–297, 1951.

[120] Fujitsu Website.
URL `http://www.fujitsu.com`

[121] Hitachi Website.
URL `http://www.hitachi.com`

[122] Maxtor Website.
URL `http://www.maxtor.com`

[123] Samsung Website.
URL `http://www.samsung.com`

[124] Seagate Website.
URL `http://www.seagate.com`

[125] Western Digital Website.
URL `http://www.wdc.com`

[126] T. Wlodek, *Developing and Managing a Large Linux Farm*, in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP 2004)*, Interlaken, Switzerland, September.

[127] T. Smith, *Managing Mature White Box Clusters at CERN*, in *Second Large Scale Cluster Computing Workshop*, Fermilab, Batavia, Illinois, USA, October 2002.
URL `http://conferences.fnal.gov/lccws/papers2/mon/LCW_White_Box.pdf`

[128] N. Talagala and D. A. Patterson, *An Analysis of Error Behaviour in a Large Storage System*, Tech. Rep. CSD-99-1042, University of California, Berkeley, February 1999.

[129] J. Lange, *Hard Drive Reliability?*, in Linux-RAID Mailing List, May 2004.
URL `http://marc.theaimsgroup.com/?l=linux-raid&m=108499785302637&w=2`

[130] M. Malhotra and K. S. Trivedi, *Reliability Analysis of Redundant Arrays of Inexpensive Disks*, in Journal of Parallel and Distributed Computing, 17(1-2):146–151, February 1993.

[131] M. Okun and A. Barak, *Atomic Writes for Data Integrity and Consistency in Shared Storage Devices for Clusters*, in Future Generation Computer Systems, 20(4):539–547, May 2004.

[132] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*, MIT Press, 2 edn., 1972.

[133] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, Interscience Series in Discrete Mathematics, Wiley, 1982.

[134] D. R. Hankerson, D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger, and J. R. Wall, *Coding Theory and Cryptography, The Essentials*, Pure and Applied Mathematics, Dekker, 2000.

[135] B. Sklar, *Reed-Solomon Codes*, April 2002.
URL `http://www.informit.com/content/images/art_sklar7_reed-solomon/elementLinks/art_sklar7_reed-solomon.pdf`

[136] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, 1995.

[137] R. B. Wells, *Applied Coding and Information Theory for Engineers*, Prentice Hall, 1999.

[138] R. Hamming, *Error Detecting and Error Correcting Codes*, in The Bell System Technical Journal, 29(2):147–160, April 1950.

[139] R. C. Bose and D. K. Ray-Chaudhuri, *On a Class of Error Correcting Binary Group Codes*, in Information and Control, 3(1):68–79, March 1960.

[140] A. Hocquenghem, *Codes Correcteurs d'Erreurs*, in Chiffres, 2:147–156, September 1959.

[141] I. S. Reed and G. Solomon, *Polynomial Codes over Certain Finite Fields*, in Journal of the Society for Industrial and Applied Mathematics, 8(2):300–304, June 1960.

[142] W. W. Peterson, *Encoding and Error-correction Procedures for the Bose-Chaudhuri Codes*, in IEEE Transactions on Information Theory, 6(4):459–470, September 1960.

[143] D. Gorenstein and N. Zierler, *A Class of Error Correcting Codes in p^m Symbols*, in Journal of the Society of Industrial and Applied Mathematics, 9(2):207–214, June 1961.

[144] E. R. Berlekamp, *Algebraic Coding Theory*, McGraw Hill, 1968.

[145] J. L. Massey, *Shift Register Synthesis and BCH Decoding*, in IEEE Transactions on Information Theory, 15(1):330–337, April 1969.

[146] S. S. Lee and M. K. Song, *An Efficient Recursive Cell Architecture of Modified Euclid's Algorithm for Decoding Reed-Solomon Codes*, in IEEE Transactions on Consumer Electronics, 48(4):845–849, March 2003.

[147] T.-K. Truong, J.-H. Jeng, and T. C. Cheng, *A New Decoding Algorithm for Correcting Both Erasures and Errors of Reed-Solomon Codes*, in IEEE Transactions on Communications, 51(3):381–388, March 2003.

[148] J. S. Plank, *A Tutorial on Reed-Solomon Coding for Fault-tolerance in RAID-like Systems*, in Software - Practice & Experience, 9(27):995–1012, September 1997.

[149] J. S. Plank and Y. Ding, *Note: Correction to the 1997 Tutorial on Reed-Solomon Coding*, in Software - Practice & Experience, 35(2):189–194, February 2005.

[150] M. Blaum, J. Brady, J. Bruck, and J. Menon, *EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures*, in IEEE Transactions on Computers, 44(2):192–202, February 1995.

[151] D. Feng, H. Jin, and J. Zhang, *Improved EVENODD Code*, in *Proceedings of 1997 IEEE International Symposium on Information Theory*, 261, Ulm, Germany, June 1997.

[152] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann, *Practical Loss-Resilient Codes*, in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 150–159, El Paso, Texas, USA, May 1997.

[153] M. Luby, M. Mitzenmacher, and M. Shokrollahi, *Analysis of Random Processes via And-Or Tree Evaluation*, in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 364– 373, San Francisco, California, USA, January 1998.

[154] R. G. Gallager, *Low-Density Parity-Check Codes*, MIT Press, 1963.

[155] M. Luby, *Benchmark Comparisons of Erasure Codes*.
URL `http://www.icsi.berkeley.edu/~luby/erasure.html`

[156] A. Rubini and J. Corbet, *Linux Device Drivers*, O'Reilly & Associates, 2 edn., June 2001.

[157] P. T. Breuer, *Private Communication*.

[158] M. Frey, *Linux Kernel Threads*.
URL `http://www.scs.ch/~frey/linux/kernelthreads.html`

[159] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 2 edn., 2002.

[160] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, *Extensible Markup Language (XML) 1.0, W3C Recommendation*, 3 edn., February 2004.

[161] Y. Matsumoto, *The Ruby Scripting Language*.
URL `http://www.ruby-lang.org/en`

[162] T. M. Steinbeck, V. Lindenstruth, and H. Tilsner, *A Control Software for the Alice High Level Trigger*, in *Computing in High Energy and Nuclear Physics Conference 2004 (CHEP 2004)*, Interlaken, Switzerland, September 2004.

[163] Kirchhoff Institute of Physics, University of Heidelberg.
URL `http://www.kip.uni-heidelberg.de`

[164] German SuSE Website.
URL `http://www.suse.de`

[165] SuSE Website (International).
URL `http://www.novell.com/linux/suse/index.html`

[166] V. Lindenstruth, T. M. Steinbeck, and A. Wiebalck, *Prozessor-Schwinger*, in Linux-Magazin, (05/03):44–47, May 2003.

[167] V. Lindenstruth, T. M. Steinbeck, and A. Wiebalck, *Fluctuating Processors*, in Linux-Magazine, (31):46–49, June 2003.

[168] Tyan Website.
URL `http://www.tyan.com`

[169] Serverworks Website.
URL `http://www.serverworks.com`

[170] 3com Website.
URL `http://www.3com.com`

[171] Trendnet Website.
URL `http://www.trendnet.com`

[172] Netgear Website.
URL `http://www.netgear.com`

[173] T. M. Steinbeck and A. Wiebalck, *MLUC: The More or Less Useful Class library*.
URL `http://www.kip.uni-heidelberg.de/ti/L3/software/`

[174] Netperf Website.
URL `http://www.netperf.org`

[175] S. Kalcher, *Messungen mit Fast- und Gigabit Ethernet*, Internship Report, Chair of Computer Science and Computer Engineering, Kirchhoff Institute of Physics, University of Heidelberg, March 2003.

[176] T. M. Steinbeck, *A Modular and Fault-Tolerant Data Transport Framework*, Ph.D. thesis, University of Heidelberg, Germany, February 2004.

[177] S. Kalcher, *Optimization of a Distributed Fault-Tolerant Mass Storage System for Clusters*, Diploma Thesis, Chair of Computer Science and Computer Engineering, Kirchhoff Institute of Physics, University of Heidelberg, December 2004.

[178] S. Tomov, M. McGuigan, R. Bennett, G. Smith, and J. Spiletic, *Benchmarking and Implementation of Probability-based Simulations on Programmable Graphics Cards*, in Computers and Graphics, 29(1), February 2005.

[179] General-purpose Computation Using Graphics Hardware.
URL `http://www.gpgpu.org`

[180] BrookGPU Website.
URL `http://www.graphics.stanford.edu/projects/brookgpu`

[181] Cg Website.
URL `http://developer.nvidia.com/page/cg_main.html`

[182] The Industry's Foundation for High Performance Graphics, *The OpenGL Graphics System: A Specification*.
URL `http://www.opengl.org`

[183] DirectX Website.
URL `http://www.microsoft.com/windows/directx`

[184] H. Tilsner, T. Alt, K. Aurbakken, G. Grastveit, H. Helstrup, V. Lindenstruth, C. Loizides, J. Nystrand, D. Roehrich, B. Skaali, T. Steinbeck, K. Ullaland, A. Vestbo, and T. Vik, *The High-Level Trigger of ALICE*, in European Physical Journal C direct, 33(S1):1041–1043, July 2004.

[185] Altera Website.
URL `http://www.altera.com`

[186] The PCI & Shared Memory Interface Library (PSI).
URL `http://www.kip.uni-heidelberg.de/ti/L3/software/`

[187] Highpoint Website.
URL `http://www.highpoint-tech.com`

[188] Debian Website.
URL `http://www.debian.org`

[189] Cisco Systems Website.
URL `http://www.cisco.com`

[190] P. Melchior, *ClusterRAID Configuration*, Chair of Computer Science and Computer Engineering, Kirchhoff Institute of Physics, Internship Report, March 2004.

[191] L. Hess and A. Wiebalck, *Blockhaus in der Ferne: Fehlertolerantes RAID mit Enhanced Network Block Devices*, in Linux-Magazin, (11/04):40–43, 2004.

Acknowledgements

There are several people who contributed to this work during the past few years in one way or the other and I would like to take this occasion to thank them for their support during this time.

First of all, I thank my supervisor Prof. Dr. Volker Lindenstruth for giving me the opportunity to work on a topic which was not in the primary research focus of his group at the time I started my PhD. Without his enthusiasm for the project, his ideas and suggestions, and the support he provided in all respects, this thesis would never have come to this point. Moreover, his open-minded attitude also allowed me to try things not right on the track of the thesis. All this made it a great time for me and I really appreciate that.

I also thank Prof. Dr. Thomas Ludwig for his interest in the project, for the invitation to workshops related to my topic, and, of course, for taking the time to act as the second referee of this thesis.

Special thanks go to Dr. Timm M. Steinbeck for all the fruitful discussions, for his help and technical advice on both architectural and implementation-related issues, and, of course, for all the fun we had during this time.

I also would like to thank Dr. Peter T. Breuer, the initiator and maintainer of the ENBD software, for his support. Debugging kernel level software from remote by analyzing my logfile snapshots and comments surely was not always a trivial task. Despite this, he always took the time to respond to the questions, comments, and problems I observed and came up quickly with solutions. His willingness to support my requests for additional features of ENBD made my life a lot easier.

Thanks also go to Sebastian Kalcher for his analyses and improvements of the ClusterRAID, which he has conducted during his time as a diploma student. His efforts and ideas helped a lot at improving the performance of the software and paved the way for a hardware-supported version.

I thank Dr. Markus Schulz and Prof. Dr. Alexander Reinefeld for their comments during the early stages of the thesis, which helped me to identify some of the crucial aspects.

Marcus Gutfleisch and Hitanshu Gandhi provided assistance with some theoretical and simulation-related aspects. Lord Hess, Holger Höbbel, and Peter Melchior helped extending the Distributed RAID from a scientific into a production-quality system.

I am also grateful to Dr. Timm Steinbeck, Dr. Heinz Tilsner, Sebastian Kalcher, and Aidan Budd for taking the time to proofread the thesis and for their valuable comments.

I am indebted to my parents and my grandparents for making all this possible and for their overall support during the time of my studies.

My deepest thanks, however, go to Nicole for her love and patience, and for always reminding me that there are more important things in life.