JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

RAIDframe & Distributed Storage

Böhm Igor
0155477
igor@bytelabs.org

Praher Jakob
0056228
jp@hapra.at

Supervised by
Dipl.-Ing. Putzinger Andreas

Institute for Information Processing and Microprocessor Technology
Johannes Kepler University Linz

Linz, June 14, 2005

Abstract

This paper is divided into two parts, both related to interesting developments in the area of computer storage. The first part deals with RAID and introduces the RAIDframe framework, a rapid tool for prototyping RAID architectures. In the second part, two distributed storage concepts, namely Storage Area Networks (SAN) and Networkd Attached Storage (NAS), are introduced and compared to Direct Attached Storage.

Over the past decade several trends in computer history have driven the design of storage subsystems towards increasing parallelism. An innovation that improves both dependability and performance of storage systems is *disk arrays* [6]. Since disk arrays consist of many disk drives and, hence, many disk arms, rather than one large drive with one disk arm, potential throughput can be increased, thus improving performance.

After having covered the basics, the need for a framework, namely RAIDframe, which enables rapid RAID prototyping will be elaborated. Many key ideas and concepts implemented in RAIDframe will be explained, in order to justify the power and flexibility of RAIDframe. Finally a section describing the most important steps which are necessary to extend the RAIDframe framework with a new architecture, shows the extensibility of the RAIDframe framework.

While storage prices keep on dropping the cost of data remains invaluable. High speed computer networks make it possible to aggregate servers into central organistation wide serverfarms. Keeping reliable storage on a per server basis can soon become a maintenance and cost problem.

Distributed storage helps organisations to aggregate reliable storage into central managed places, thus reducing the amount of per server maintenance overhead. Currently there exist two approaches for doing distributed storage. Storage Area Networks and Network Attached Storage. This paper introduces the main concepts behind them, and compares distributed to local storage. Two concrete distributed storage protocols are presented to help the reader in getting a better overview.

Table of Contents

Chapter 1

Introduction

1.1 Structure of this Paper

The first part of this paper introduces the RAIDframe framework. In order to fully understand all the RAID specific topics covered by this paper, it is necessary to start with the basics of RAID[1] systems. Thus an overview of RAID categeries, combined with the analysis of advantages and disadvantages of certain RAID levels, is given first.

After having covered the basics, the need for a framework, namely RAIDframe, which enables rapid RAID prototyping will be elaborated. Many key ideas and concepts implemented in RAIDframe will be explained, in order to justify the power and flexibility of RAIDframe. Finally a section describing the most important steps which are necessary to extend the RAIDframe framework with a new architecture, shows the extendability of the RAIDframe framework.

In the second part of the paper we discuss distributed storage. We will first introduce the major flavors of distributed storage technology, namely Storage Area Networks (SAN) and Network Attached Storage (NAS) and compare them to Direct Attached Storage (DAS). After that we will discuss distributed storage over TCP/IP networks. Here we do a comparison of the technologies - iSCSI as an exponent of SAN technology and the NFS protocol suite as an exponent of NAS. We conclude by showing differences and performance impacts of the two approaches.

[1]RAID - Redundant Arrays of Inexpensive Disks

1.2 Motivation

Over the past decade several trends in computer history have driven the design of storage subsystems towards increasing parallelism. An innovation that improves both dependability and performance of storage systems is *disk arrays* [6]. Since disk arrays consist of many disk drives and, hence, many disk arms, rather than one large drive with one disk arm, potential throughput can be increased, thus improving performance.

Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability can be improved by adding redundant disks to the array to tolerate faults. This implies that if a single disk fails, the lost information can be reconstructed from redundant information

There are several approaches for maintaining redundant data, and in 1987 these different approaches were categorized into a taxonomy known as RAID by a research group at U.C. Berkeley headed by David A. Patterson.

While storage prices keep on dropping the cost of data remains invaluable. High speed computer networks make it possible to aggreate servers into central, organistation wide serverfarms. Keeping reliable storage on a per server basis can soon become a maintenance and cost problem.

Distributed storage helps organisations to aggregate reliable storage into central managed places thus reducing the amount of per server maintenance overhead. Currently there exist two approaches for doing distributed storage. Storage Area Networks and Network Attached Storage. This paper introduces the main concepts behind them and compares distributed to local storage. Two concrete distributed storage protocols are presented to help the reader in getting a better overview.

Chapter 2

RAID Levels

There is a numerical classification of RAID which divides RAID systems into different levels based on their fault tolerance, and the overhead in redundant disks.

2.1 RAID 0 - No Redundancy

This RAID level refers to a nonredundant disk array, indicating the data are striped across several disks but without redundancy to tolerate disk failure (see figure 2.1). By striping across a set of disks, storage management can be simplified since a collection of disks appears as a single large disk to an application.

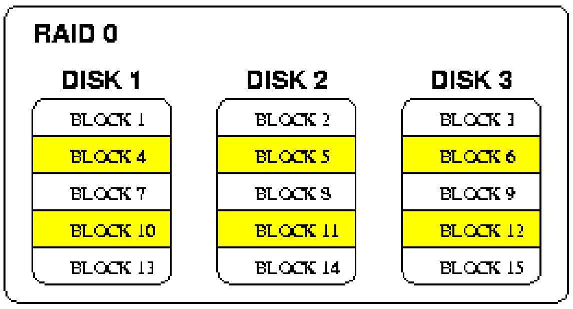

Figure 2.1: RAID 0 - No redundancy.

Another advantage of striping across a set of disks is an improvement in performance for large disk accesses, since many disks can operate at once. Thus non-redundant disk arrays are widely used in super-computing environments where performance and capacity, rather than reliability, are the primary concerns.

2.2 RAID 1 - Mirroring

RAID 1 is the traditional scheme for tolerating disk failure, also called mirroring or shadowing (see figure 2.2).

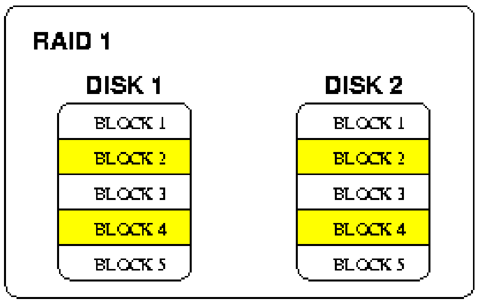

Figure 2.2: RAID 1 - Mirroring.

This approach uses twice as many disks as RAID 0 because whenever data are written to one disk, those data are also written to a redundant disk, thus there are always two copies of the information leading to increased reliability by a factor of two. Since reliability is the linear multiple of the number of member disks, it can easily be increased by creating more than two copies of data.

RAID level 1 defines the user data to be block-striped across the mirror pairs. Traditional mirrored systems rather fill each disk with consecutive user data before switching to the next disk, which can be thought of as setting the stripe unit to the size of one disk.

2.3 RAID 3 - Bit-Interleaved Parity

Having a complete copy of the original data for each disk is a very expensive solution. The cost of higher availability can be decreased by only adding enough redundant information, referred to as *parity*, to restore the lost information on a failure.

In a RAID level 3 setup, data is conceptually interleaved bit-wise over the data disks, with an additional parity disk tolerating a single disk failure (see figure 2.3). With this setup each read request needs to access all data disks and each write request needs to access all data disks and the parity disk. Thus it is only possible to serve one read or write request at a time.

If an error occurs during a read operation, the corresponding disk controller reports a read data error so the RAID system knows which disk has failed,

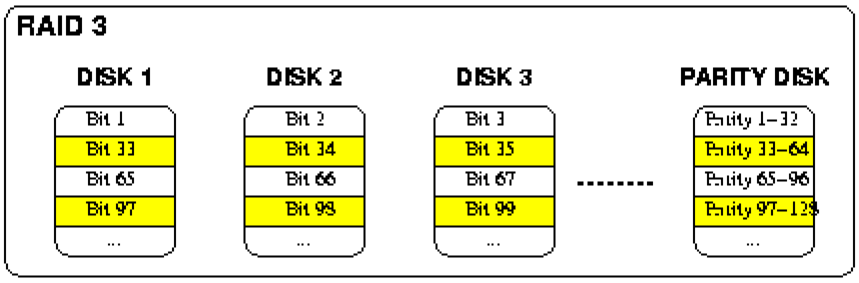

Figure 2.3: RAID 3 - Bit-Interleaved Parity.

and thus is able to reconstruct the missing data by `XOR`*ing* all of the remaining data disks plus the parity disk. If we assume that disk 3 in the RAID level 3 diagram in figure 2.3 has failed, the reconstructions would work like the follwing example, where B denotes *Bit*, and P denotes *Parity*:

$$(P_{1-32} = B_1 \oplus B_2 \oplus B_3 \oplus \cdots \oplus B_{32}) \rightarrow (B_3 = B_1 \oplus B_2 \oplus P_{1-32} \oplus \cdots \oplus B_{32})$$

2.4 RAID 4 and 5 - Block-Interleaved Parity and Distributed Block-Interleaved Parity

A disadvantage of RAID 3 is that every disk access goes to all disks, but many of todays applications would prefer to issue smaller disk accesses which should occur in parallel if they are independent. That is the purpose of RAID levels 4 and 5.

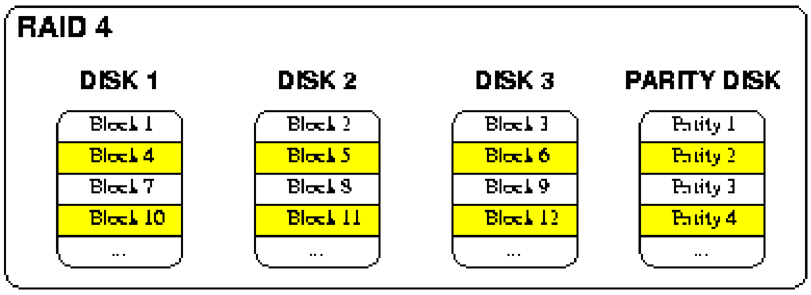

Figure 2.4: RAID 4 - Block-interleaved Parity.

Both, RAID 4 and 5, use the same ratio of data and check disks as RAID 3, but they access data differently [6]. The main advantage of block-interleaved parity disk arrays is that the parity is stored as blocks and associated with a set of data blocks (see figure 2.4). The size of these blocks is called the striping unit. Read requests smaller than the striping unit access only a single data disk. Write requests must update the corresponding data blocks and recalculate and update the parity blocks.

Because in a RAID 4 setup there is only one parity disk which must be updated on every write operation, the parity disk will soon become a bottleneck for write intensive applications. RAID level 5 eliminates the parity disk bottleneck by distributing the parity blocks uniformly over all disks (see figure 2.5).

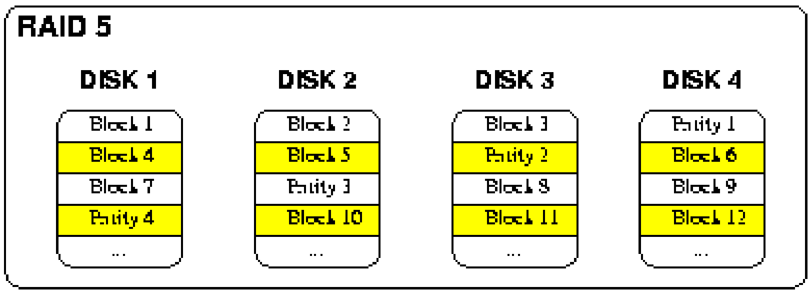

Figure 2.5: RAID 5 - Distributed Block-interleaved Parity.

The organization of RAID 5 clearly shows that the parity associated with each row of data blocks is no longer restricted to a single disk. This allows multiple writes to occur simultaneously as long as the data and parity blocks are not located in the same disks.

In RAID Level 5, there are a variety of ways to lay out data and parity such that parity is evenly distributed over the disks [14]. The structure shown in Figure 2.5 is called the *left-symmetric* organization and is formed by first placing the parity units along the diagonal and then placing the consecutive user data units on consecutive disks at the lowest available offset on each disk. This method for assigning data units to disks assures that, if there are any accesses in the workload large enough to span many stripe units, the maximum possible number of disks will be used to service them [8].

2.5 RAID 6 - P+Q Redundancy

The parity based RAID levels 1 through 5 only protect against a single disk failure. For some mission critical applications this may not be sufficient, thus parity can be generalized to have a second calculation over the data blocks combined with a second parity block. This second parity block would allow for recovery in case of a second disk failure.

P+Q Redundancy usually uses block-level striping across a set of drives, just like in RAID 5, and a second set of parity is calculated and written across all the drives (see figure 2.6).

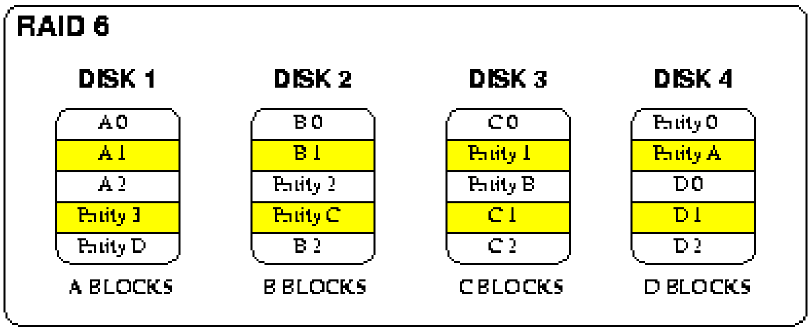

Figure 2.6: RAID 6 - P+Q Redundancy.

2.6 Nested RAID Levels

Until now we have assumed that each RAID solution consists of a set of physical hard disks as its basic elements. We want to extend this assumption to the point where, one RAID level can use another RAID level as its basic element. Such a nested RAID array resembles a tree like structure were all nodes represent a RAID level, and the leafs represent the physical disks at the bottom (see figure 2.7).

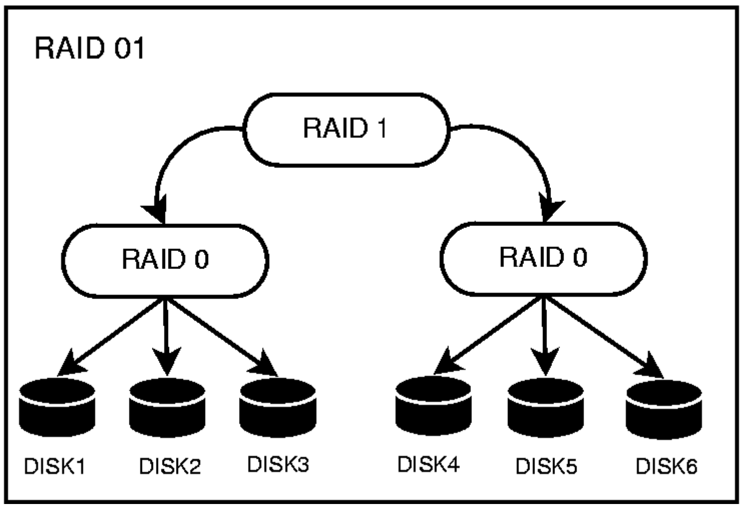

Figure 2.7: RAID 01 - RAID 1 consisting of RAID 0 arrays.

Nested RAID arrays are usually denoted by joining the numbers indicating the RAID levels into a single number. An example for this taxonomy would be the combination of multiple RAID 0 arrays stored on physical disks, with a RAID 1 on top, referred to as RAID 01 (mirror of stripes).

One reason to nest RAID levels is to increase performance and redundancy due to a combination of a RAID type which provides redundancy (e.g. RAID 1), with a RAID type that boosts performance (e.g. RAID 0).

Chapter 3

RAIDFrame

3.1 General Concepts

In the previous chapter we discussed the structure and operation of disk arrays, explaining the different data layouts and fault tolerance for each of the original RAID levels. It should have become quite obvious how complex array software used to control the disks can get. Unfortunately, using the traditional manual firmware-design approach employed by storage system designers, implementing control software for these redundant disk array architectures has led to long product-development times and uncertain product reliability [13].

Since almost no code is shared between RAID implementations which follow the traditional development approach, the software becomes overly complex. It is a very hard and tedious task to test such complex software, thus providing a framework for efficient and rapid development of array software for a particular situation which performs optimally, is the main goal of RAIDframe.

To do this, the RAIDframe project has aimed to increase the amount of code reused between RAID designs. Thus things like generalized error-recovery mechanisms and means for verifying the correctness of a design before it is even implemented, become possible.

3.1.1 Identifying a Common Set of Primitive RAID Operations

Despite the complexity of various RAID levels, a typical raid controller usually maps all RAID operations to a relatively small set of corresponding disk operations, refered to as *primitive operations.* In general, such primitive operations consist of routines for disk access, redundancy calculation and resource allocation.

Such operations can be viewed as instructions of a RAID architecture, with various constraints imposed upon the sequence of execution of such instructions. We can use these instructions to construct RAID operations. Just as computer architects use the instructions of a certain instruction set architecture to implement various programs, it is possible to use our RAID instructions set to create RAID operations which can be treated as programs.

By only using RAID instructions which are present in the RAID instruction set, much more code can be shared across various RAID level implementations. When designing an efficient RAID instructions, it is important to create modular instructions, which only change orthogonally if the architecture changes.

The following incomplete listing of common primitive RAID operations will suffice to explain the key ideas of the RAIDframe architecture:

- `Rd:` copy data from disk to buffer
- `Wd:` copy data from buffer to disk
- `MemA:` acquire a buffer
- `MemD:` release a buffer
- `XOR:` xor contents of buffers
- ...

3.1.2 Building RAID Operations Based on a Set of Primitive Operations

It has already been mentioned that there are several constraints imposed upon the sequence of execution of RAID instructions. The order in which such primitive operations are executed is solely a function of the data and control dependencies which exist between them [8]. Similar to a dynamically scheduled processor, which reorders the execution of instructions as long als there are no dependencies between them, in order to exploit instruction level parallelism, a RAID array designer must know the necessary dependencies which exist between primitive RAID operations in order to efficiently implement them.

3.2 Using Graphs for RAID Access Sequence Specification

In order to model RAID operations, *directed, acyclic graphs*, which we will referr to as DAGs, are used. By using DAGs to model RAID operations consisting of primitive RAID operations, it is guaranteed that the set of primitive operations has a strict partial order. In other words, for all a, b, and c in the set of primitive RAID operations, we have that:

- $\neg(aRa)$ *(irreflexivity)*
- if aRb then $\neg(bRa)$ *(asymmetry)*
- if aRb and bRc then aRc *(transitivity)*

Thus every DAG representation of a RAID operation contains all ordering constraints which bind the primitive operations together.

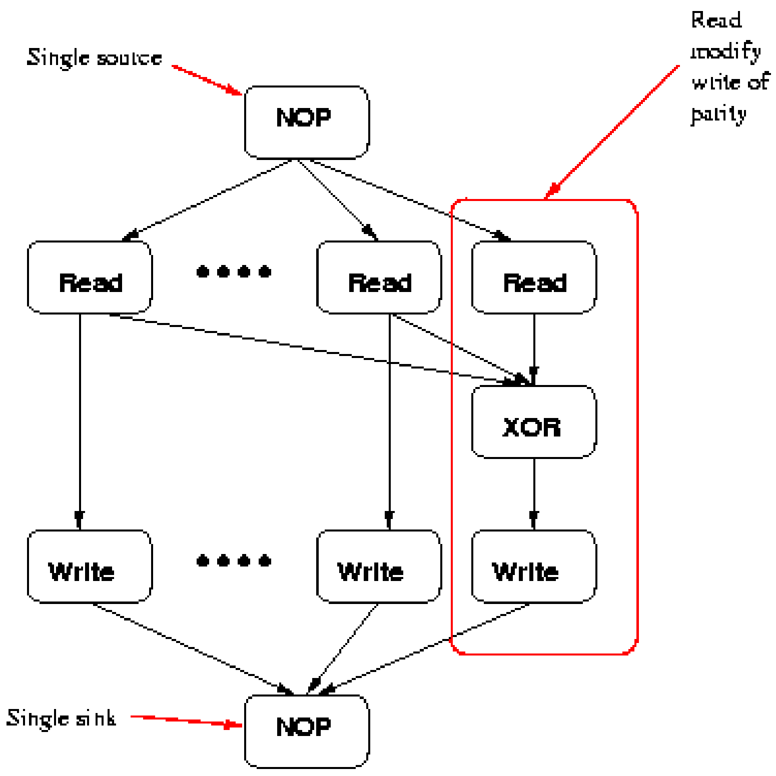

Figure 3.1: DAG - Small RAID 4/5 write operation representation.

Figure 3.1 illustrates a DAG which shows a small RAID 4/5 write operation. A node represents a primitive RAID operation and the directed arcs represent data or control dependencies between primitive operations. The `Read-XOR` arcs for example imply a data dependency since parity is computed from the old data. If a graph does not contain a single source or sink node, an extra `NOP` node is added which has no effect upon the RAID array operation represented by the graph.

3.2.1 Graph Representation of Error Recovery Strategies

Since RAID operations also need to be able to cope with errors, it is necessary to implement some type of error recovery. Basically there are two possible approaches to error recovery, namely *foward* error recovery and *backward* error recovery.

Forward error recovery requires anticipating all possible errors and manually coding all actions for completing operations once an error occured [8]. This approach usually requires a lot of code and is hard to modify once it has been layed out to handle a set of errors appropriately. *Backward* error recovery is an approach which is mainly used in database systems supporting transactions. The concept of transactions allows the composition of undoable atomic database operations. Thus if an error occurs in the middle of a transaction, the system undoes all effects of that transaction and provides the illusion that the transaction never occured. In order to make this error recovery approach feasible, it is necessary for the underlying database system to detect and recover from errors at the cost of keeping logs of all operations which happen during the transition of one consistant state into another.

RAIDframe implements a hybrid approach called *roll-away* error recovery. This approach is incorporated within DAGs and it basically combines foward error recovery without the need of accounting for all possible error scenarios, and, when necessary, it uses backward error recovery without the cost of logging state information [8]. If possible, RAID operations are split into two phases by introducing a *commit* barrier to the DAG, where phase one contains all operations which access data without modyfing it, and phase two consists of all operations which actually modify symbols on disk. Therefore a `Commit` node is added to the DAG to distinguish between these two phases.

The small RAID 4/5 write example can be split into two phases by intruducing such a `Commit` barrier as shown in figure 3.2. Phase one would contain all primitive RAID operations which can be undone easily like disk `Read` operations and the parity `XOR` calculation, and phase two contains all operations which modify data words like the actual update of parity and data information. Thus the *Commit* node prevents writes of new data from proceeding until all reads of old data and the computation of parity have completed. One can think of such *Commit* nodes as the sink of all read operations and the source of all `Write` operations.

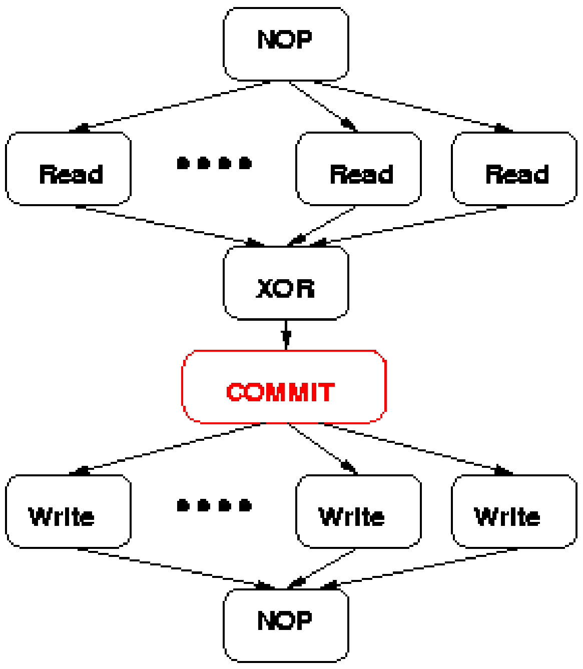

Figure 3.2: DAG - RAID operation representation with Commit barrier.

Going even further, we can think of these DAGs as state machines, thus all methods of automated model checking can also be applied to DAGs, which means that it is possible to verify RAID designs before implementing them.

3.3 RAIDframe Features

The original purpose of RAIDframe was to provide an environment where RAID experiments could easily be performed, and where new RAID algorithms could easily be implemented and tested. As distributed by CMU[1], RAIDframe consisted of a RAID simulator, a user-land disk driver, and a kernel-level device driver for (then) Digital Unix. RAIDframe, as found in various BSD flavors today, is a fully-integrated kernel-level device driver supporting many features such as:

- Hot Spares - these are disks which are on-line, but are not actively used in an existing file system. Should a disk fail, the raid driver is capable of reconstructing the failed disk on to the hot spare.

[1] `CMU` - Carnegie Mellon University

- Component Labels - contain important information about the component such as a user-specified serial number, the row and column of that component in the RAID set, and whether the data and parity on the component is clean. If the RAIDframe driver detects that component labels are very inconsistant (e.g. the serial numbers do not match or the label is not consistant with the assigned place in the set), the device will fail to configure.
- Root on RAID - means that a RAID filesystem can be used as the root filesystem.

RAIDframe provides a number of different RAID levels including basic RAID architectures as well as a number of experimental architectures:

- RAID 0 - provides simple data striping across the components.
- RAID 1 - provides mirroring.
- RAID 4 - provides data striping across the components, with parity stored on a dedicated drive (in this case, the last component).
- RAID 5 - provides data striping across the components, with parity distributed across all the components.
- Even-Odd parity
- Chained declustering - the primary data copy in disk i, has backup copy on $disk(i+1)\ mod\ n$, where n is the total number of disks employed.
- Interleaved declustering - the backup copy of the primary data of a disk is broken up into multiple subpartitions where each of the subpartitions is stored on a different disk within the same disk array [2].

3.4 Internal Architecture

RAIDframe provides extensibility through separation of architectural policy from execution mechanism [3] by splitting the internal architecture up into several modules which seperate stable infrastructural code from user modifiable library code.

Figure 3.3 displays all major RAIDframe modules:

- `State Machine:` A central state machine is responsible for processing user requests by creating graphs and submitting them for execution.

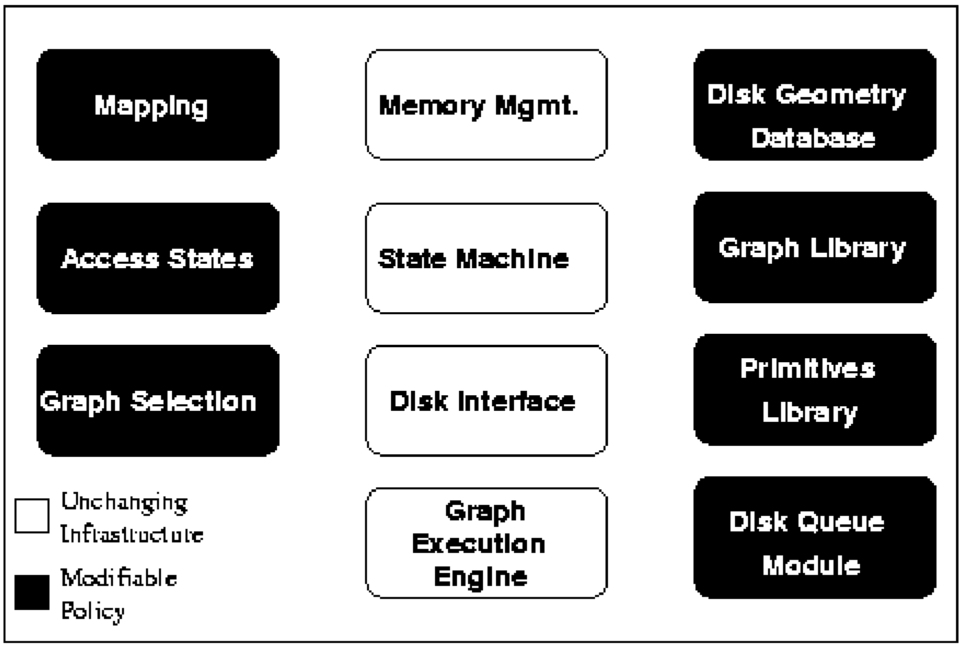

Figure 3.3: RAIDframe - Infrastructure and Modules.

For user initiated disk access requests and the reconstruction process, most RAID architectures use a state machine which is illustrated in figure 3.4.

Most RAID architectures use a state machine which is similar to the left hand side of figure 3.4.

- `Graph Execution Engine:` This engine is the primary infrastructure module of RAIDframe and it is solely responsible for executing a DAG, while trying to exploit a maximum level of parallelism when executing primitive RAID operations. Since a DAG is a very general Datastructure, the Graph Execution Engine does not need to have any knowledge about the RAID architecture represented by the DAG.

- `Disk Interface:` This module organizes pending disk operations based on queing disciplines specified at the time of configuration.

- `Disk Queue Module:` RAIDframe allows to queue disk requests either directly at the disks or within RAIDframe itself. If queueing within RAIDframe is selected, multiple queueing policies like First In First Out (FIFO), Shortest Seek Time First (SSTF) etc. are available.

- `Disk Geometry Database:` This database contains disk parameters like tracks per cylinder, as well as perfomance parameters like seek time or rpm, for the disk simulator.

- `Mapping:` Before any block ranges are locked in the disk array, all access goes through this module which invokes RAID architecture specific mapping routines which are able to map the correct sectors and parity units for a given RAID address.
- `Graph Selection:` Each RAID architecture requires a graph selection algorithm which selects the appropriate DAG from the graph library for a specific user request, given the current state of the RAID array.
- `Graph Library:` This library includes functions which are capable of creating DAGs when called by the graph selection engine.
- `Primitive Operations Library:` These functions implement the primitive RAID operations instruction set such as `XOR` or `DiskRD` and thus abstract single device operations.

3.5 Reconstruction Architecture

3.5.1 Reconstruction Algorithm

A reconstruction algorithm is a strategy used by a background reconstruction process to regenerate data that existed on the failed disk and store it on a replacement disk [7]. RAIDframe uses a *disk-oriented* instead of a *stripe-oriented* reconstruction algorithm as described in [7], mainly because the *disk-oriented* algorithm performs much better at consistently utilizing all disk bandwidth not absorbed by user access for disk reconstruction. Instead of creating only one reconstruction process as it is done with the *stripe-oriented* approach, C reconstruction processes are created, where $C - 1$ processes are associated with the surviving disks, and the remaining process is associated with the replacement disk. The algorithm for the surviving disks is as follows:

- `repeat`
    1. Find the lowest numbered unit on `this` disk that is needed for reconstruction.
    2. Read the indicated unit into a buffer.
    3. Wait for the read to complete.
    4. Submit the unit's data to a centralized buffer manager for further processing.
- `until` *(all necessary units have been read)*

The replacement disk implements the following algorithm:

- `repeat`
  1. Request a buffer of reconstructed data from a centralized buffer manager.
  2. Issue a write of the buffer to the replacement disk.
  3. Wait for the write to complete.
- `until` *(failed disk has been reconstructed)*

3.5.2 Reconstruction State Machine

Whenever a disk fails, a seperate state machine which is responsible for the reconstruction process is initiated, and works in parallel with the state machine which is responsible for user initiated disk access requests (see figure 3.4).

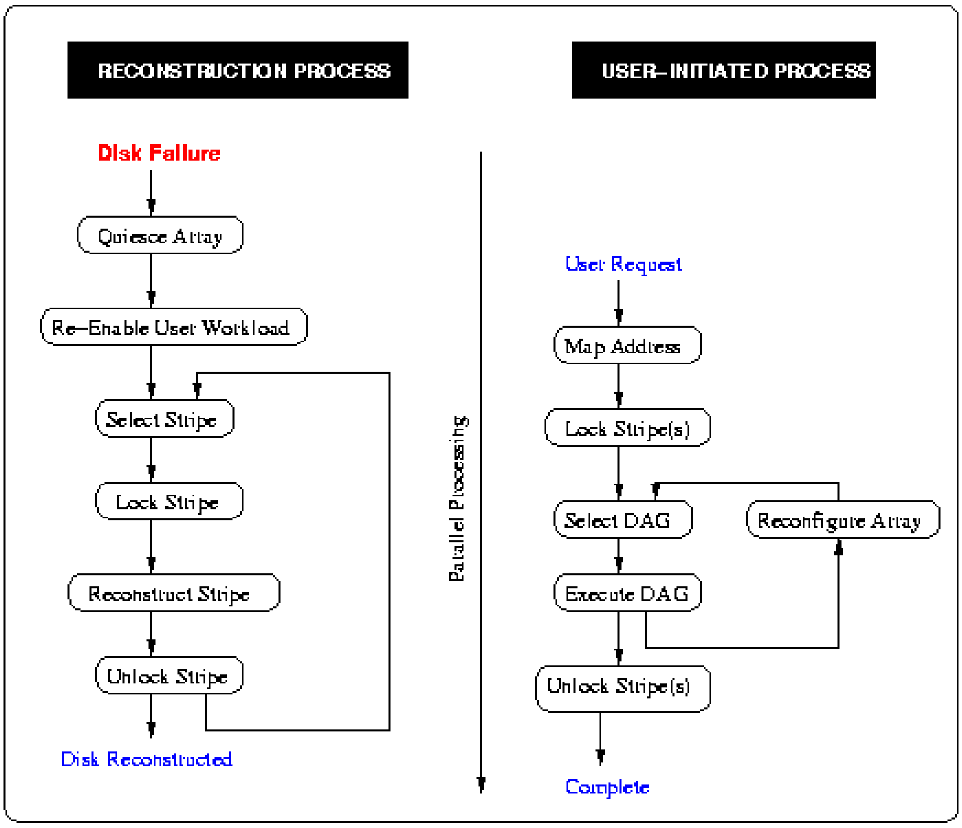

Figure 3.4: RAIDframe - State machines.

Of course the reconstruction process has a lower priority than user-initiated access requests. It dispatches reconstruction disk accesses in a batch style manner until all data on the failed disk has been restored.

3.6 Extensibility of RAIDframe

After this coarse architectural overview and insights into the key ideas of RAIDframe, it would also be interesting to look at what actually needs to be done to add a new RAID architecture to the Framework. Obviously this task should not be that hard because of the power and flexibility which RAIDframe provides.

The following listing describes major steps which are necessary in order to extend the RAIDframe framework with a new architecture:

- `Register Architecture:` First of all it is necessary to register the new architecture and implement a layout-specific parsing routine. Thus RAIDframe will be able to call the appropriate parsing callback function upon discovering the use of the new architecture in the configuration file.
- `Configure and Initialize Architecture:` One must also hook up array- and layout-specific start-of-day configuration and initialization functions in order to be able to allocate any extra resources the new RAID architecture needs.
- `Define Mapping Functions:` Since the raid device address space which get's exported to the host is only a logical representation, it is necessary to map these logical addresses and parity locations to the correct physical addresses in the RAID array (see figure 3.5). The two most important mapping functions are:
  - `MapSector:` This function implements the mapping of logical sectors to physical disk array sectors. Each array architecture must implement such a mapping function which must yield a unique mapping to the physical disks.
  - `MapParity:` This function is similar to the `MapSector` function except that it maps a logical parity sector to a physical parity sector.
- `Define Stripe Identification Function:` This function identifies which physical disks contain sectors in a particular stripe.

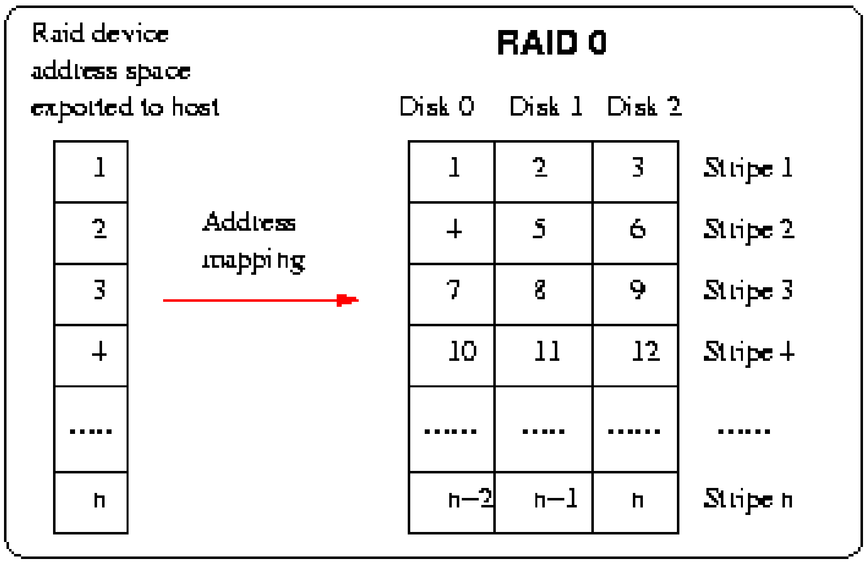

Figure 3.5: RAIDframe - Logical to physical address mapping.

- `Implement DAG Selection Functions:` Whenever an I/O request enters the RAID system it is passed through this function which is basically responsible to choose the appropriate DAG create function for this particular RAID access type.

The above steps represent the most important steps when implementing a new RAID architecture. It would be a tremendous and tedious task to design and test a new RAID architecture without this framework. The most important advantage of RAIDframe though is, that very well tested code can be reused at these low level layers.

Chapter 4

Distributed Storage

4.1 Overview

The most common setup for computer storage systems is to use the so called Direct Attached Storage (DAS), also referred to as local storage. DAS refers to storage that is directly connected to the local bus system of the computer. Some common buses for local storage access are Small Computer System Interconnect (SCSI) or Parallel Advanced Technology Attachement (P-ATA), nominally refered to as IDE, and more recently, Serial Advanced Technology Attachement (S-ATA).

DAS has several limitations. The biggest one is that every host has its own private storage system. One might think that because of the low prices per storage unit this wouldn't matter these days. But looking closer at the value of a computer system, the data of even a small company is quickly worth more than EUR 1000000. So storage is cheap but data is costly. The idea behind distributed storage is to use synergies between individual computers, thus minimizing storage administration and lowering the much cited Total Cost of Ownership (TCO).

Distributed storage systems combine networking technology with storage technology. There are several different systems in use today, which can be classified based on several features, like:

- Use of existing infrastructure vs. new infrastructure.
  Does the storage system use existing networking hardware, like Twisted Pair cabling, Ethernet Network Adapters and Switches, or does it operate on completely new networking hard- and software?
- Separated storage systems vs. combined storage and data networks.
  Is it possible to plug in a storage system next to an ordinary network

server and use it over the same network? This implies that existing network technologies are used.

- Purely software based systems vs. hybrid systems.
- Data Access format.
  How is data accessed over the network? - In raw block form (logical) or based on a networked file systems.
- Concurrency.
  Is it possible to access the storage node from more than one system at the same time?

This section will discuss the following common distributed storage technologies:

- `Storage Area Network (SAN):`
  The concept of SANs is more or less a networked version of DAS. Instead of having a point-to-point connection between a host and a storage device, several hosts can access storage devices on an often separated network. The protocol for SAN is like block oriented DAS.
- `Network Attached Storage (NAS):`
  Operates on existing networking technology. NAS uses the concept of network filesystems, where the server exports files/filesystems based on a specific protocol. NAS servers are more or less file servers which hide the storage management from the other nodes. NAS servers are data access file oriented, which implies that both the data access and the metadata access have to rely on the server.

The rest of this chapter will introduce three different storage concepts, starting with Direct Attached Storage followed by a discussion about Storage Area Networks and Network Attached Storage.

In the next chapter we will look at Distributed Storage over TCP/IP networks. First we introduce iSCSI as a SAN technology over TCP/IP, and then we take a closer look at NFS as an interesting NAS protocol, that can be used over IP networks. At the end of this chapter we will see a performance evaluation between these two protocols.

4.2 Local Storage: Direct Attached Storage

Direct Attached Storage (DAS) is used in most computer systems today. In DAS every computer has it's own local storage system. Since we will often refer to DAS in this paper, we explore the details of a typical DAS setup in greater detail now.

DAS is also called local or non networked storage. Typical server environments use a Small Computer System Interface (SCSI) and nowadays also Serial Advanced Technology Attachment (S-ATA) environment to connect to local storage.

A typical SCSI setup consists of:

- Host Bus Adapter (HBA) attached to the computer
- SCSI controller on every device

The HBA is directly plugged into one of the computers internal IO buses like PCI, PCMCIA, ... The operating system driver knows how to talk to the HBA and the SCSI HBA talks to the storage devices on the SCSI bus via the SCSI protocol. For the storage devices (eg. hard disks) to understand the SCSI protocol, it has to have an embedded SCSI controller which converts the SCSI commands to device specific actions.

The data access method is very low level. It is based on a block granularity level. Blocks are typically multiple physical sectors and it is the duty of the user (e.g. the operating system) to keep track (a) to which storage device on the SCSI (or ATA) bus it wants to talk to and (b) which sectors it wants to access.

Addressing of SCSI devices typcially involves the following information:

- SCSI adpater (HBA) number (host)
- SCSI channel number (bus)
- SCSI ID number (target)
- Logical Unit Number (LUN)

Each HBA may control one or more SCSI buses, each SCSI bus can have multiple devices attached to it. The HBA also shows up as a SCSI device and is called the initiator. Each SCSI device has exactly one SCSI ID. But a SCSI device can have multiple Logical Unit Numbers (LUNs). LUNs are important for storage devices which have more than one storage unit, for instance sophisticated tapes or CD changers which can talk to multiple CDs simultaneously. So the addressing path for a device unit in a SCSI setup is the path $< host, bus, target, LUN >$.

Addressing sectors on a storage device is also abstracted by generic geometry information:

- Cylinder (one track on all platters)
- Heads
- Sectors

The smallest addressable unit of a storage device is often called a sector. The size of a disk is made of $sectors * cyclinders * heads$. In the past, this was really the way to address data on disks, but today this is only a high level addressing mode which gets translated to the actual physical information by the storage disk. In fact other storage devices, like flash disks or USB sticks, often use these addressing modes.

So the total path of a sector in a typical SCSI setup looks like:
$< host, bus, target, LUN > . < cylinder, head, sector >$.

4.3 Storage Area Networks (SAN)

Storage Area Networks (SANs) are basically Direct Allocated Storage (DAS) devices with a longer wire. This means that they also use a block oriented raw access to storage devices, like ATA or SCSI devices in DAS. Typically SANs also use their own network infrastructure (hard- and software) to access the SAN devices. These two factors, distinguish SANs from its direct counterpart, the NAS technology, which we discuss in the next section. NAS works through a networking filesystem and uses an already existing network infrastructure.

SAN can be thought of as an extension of DAS, where DAS is a point-to-point link between the storage and the server (each server has its own DAS infrastructure), and a SAN allows many computers to access many storage devices over a shared network.

Most SANs in use today are based on Fibre Channel Technology. Fibre Channel is a whole network stack for performing serial network communication. Fibre Channel was developed for supercomputer communication and is nowadays most often used in SANs.

There are 3 Fibre Channel topologies:

- Point-to-Point: Two devices are connected back to back.
- Arbitrated Loop: All devices are in a loop or ring, similar to the token ring topology.
- Switched Fabric: All devices are connected to Fibre Channel switches, similar to modern Ethernet implementations.

Fibre Channel can also be combined with other higher layer protocols, like IP or even SCSI over Fibre Channel. Fibre Channel currently supports speeds at 1 GBit/s and 2 GBit/s while products at higher speeds are being developed.

A SAN works like a distributed DAS. Each computer (or host) participating in a SAN has one or more SAN HBAs. The host itself is referred to as the initiator as opposed to DAS where the HBA is the initiator. Another difference is that in a DAS setup there is always exactly one path to the storage system. In a SAN setup there can be multiple paths to the same storage device.

These new concepts also changed the way operating systems see storage devices. Operating systems have to adapt from the static view of storage devices, which get discovered at startup, to a distributed view. With DAS, pluggability was often impossible. Adding, changing or removing a storage device usually meant that the DAS had to be disabled or the computer had to be restarted.

SANs on the other hand add much more powerful ways to discover and maintain storage devices. For instance SAN devices can be mulitpathed, the operating system can not rely that the path to the device will stay the same over time, and thus uniquely identifies the storage device.

This makes it important to distinguish a specific path and the corresponding device more clearly. When a device is first discovered by the SAN software, a logical device has to be created. After that, every path pointing to that device has to be combined under that logical device. Also the logical device has to be stored persistently, so that the device names stay consistent over reboots or even if all the paths get exchanged.

The traditional approach where the operating system deals exclusively with a storage device, makes it impossible to share such devices without special file systems. Thus, in order for multiple nodes to be able to use a shared SAN device, other file systems, also called SAN filesystems or global filesystems, have to be employed. These filesystems store additional information that make them suitable for concurrent access. We will look into SAN filesystems in greater detail when we discuss the Global Filesystem.

4.4 Network Attached Storage (NAS)

The term Network Attached Storage (NAS) is quite ambiguous. Taken literally it just means that you attach a storage device into your network and access that storage through some network communication. NAS is shared storage. In practice NAS is commonly just a dedicated file server that can be accessed via a network file system.

Compared to SAN, NAS technology is not something new. One of the earliest network applications was to access files stored on remote computers. TFTP and later FTP were one of the first uses of the ArpaNet and later the InterNet. Network Attached Storage (NAS) isn't something revolutionary. A NAS provider is more ore less a dedicated file server which aggregates all the data of a computer network.

As already mentioned, NAS is not revolutionary. The NAS systems themselves are most likely built based on computer systems with DAS or even SAN storage devices. The most important difference between SAN and NAS is the way data is exposed and accessed. SAN clients access data as an array of blocks. A NAS client on the other hand, accesses data as files in one or more file systems. The first gets a so called handle to a data and then manipulates the files through that handle.

A *file system* is an implementation of a file name space containing files. It provides the basis for administration and space allocation. A file is a single named object consisting of data and attributes, residing in a file system. The term regular file is often used to refer to a simple byte stream, not a directory or symbolic link. A file handle uniquely identifies a file on a server.

So a NAS device operates at a much higher level. The NAS server abstracts from the raw block based access. Data access is based on files in a file system. Thus the NAS protocol has to support much more concepts:

- File Manipulation
- Directory Manipulation
- Concurrency Control (locking)
- Access Control
- Advanced Topics

File Manipulation is all abouting creating, reading, writing and deleting files. Directories are a way to hierarchically store files. Directory manipulations deal with creating and deleting directories. Access control deals with who is allowed to access certain files or directories. This is often referred to as discretionary access control.

So instead of just reading and writing sectors or blocks at some geometric location, a NAS storage device has to deal with many high level concepts. For instance:

- What is a valid file name?
- What is a valid directory?
- What is a valid path?
- Which transport protocol is used?
- What form of concurrency is allowed?
- How is a user identified?

There are many different network filesystems. The two most wide spread protocols in use today are the Network File System (NFS), and the Common Internet File System (CIFS). NFS is a very popular Unix file system originally implemented by SUN Microsystems. CIFS on the other hand is the official name for the Microsoft Windows(TM) network filesystem.

Most NAS server appliances support multiple file systems. So you can choose the network filesystem that fits best to your operating system.

A very critical point for NAS systems is data throughput and I/O performance. The best NAS system is useless, when it can't scale to the needs of the NAS clients. We will explore this even further after having introduced the other distributed storage concepts.

Chapter 5

Distributed Storage over TCP/IP

In this chapter we will look at concrete distributed storage protocols which are specified to work over existing TCP/IP networks.

The SAN protocol over TCP/IP is the so called Internet SCSI (iSCSI). As a NAS protocol we will discuss the NFS protocol in its latest release, which provides special enhancements for working better over existing internet infrastructures.

We conclude this chapter by further comparing these protocols and evaluating the performance of NAS and SAN based protocols.

5.1 SAN over TCP/IP - iSCSI

Internet SCSI (iSCSI) is a SAN technology with a special focus on existing infrastructure. By taking the best technology from networking and DAS, it uses the SCSI command set over the TCP/IP protocol.

Being a typical SAN protocol, it works on blocks (physical addressing). An iSCSI client is called an initiator and an iSCSI storage device is called a target.

Instead of a HBA the iSCSI initiator uses an ordinary Network Interface Card (NIC). The iSCSI initiator communicates with a storage router or gateway, which knows how to map the iSCSI commands to physical storage.

As can be seen in figure 5.1, iSCSI uses existing network infrastructures. Instead of having to buy an expensive Fibre Channel HBA and switches,

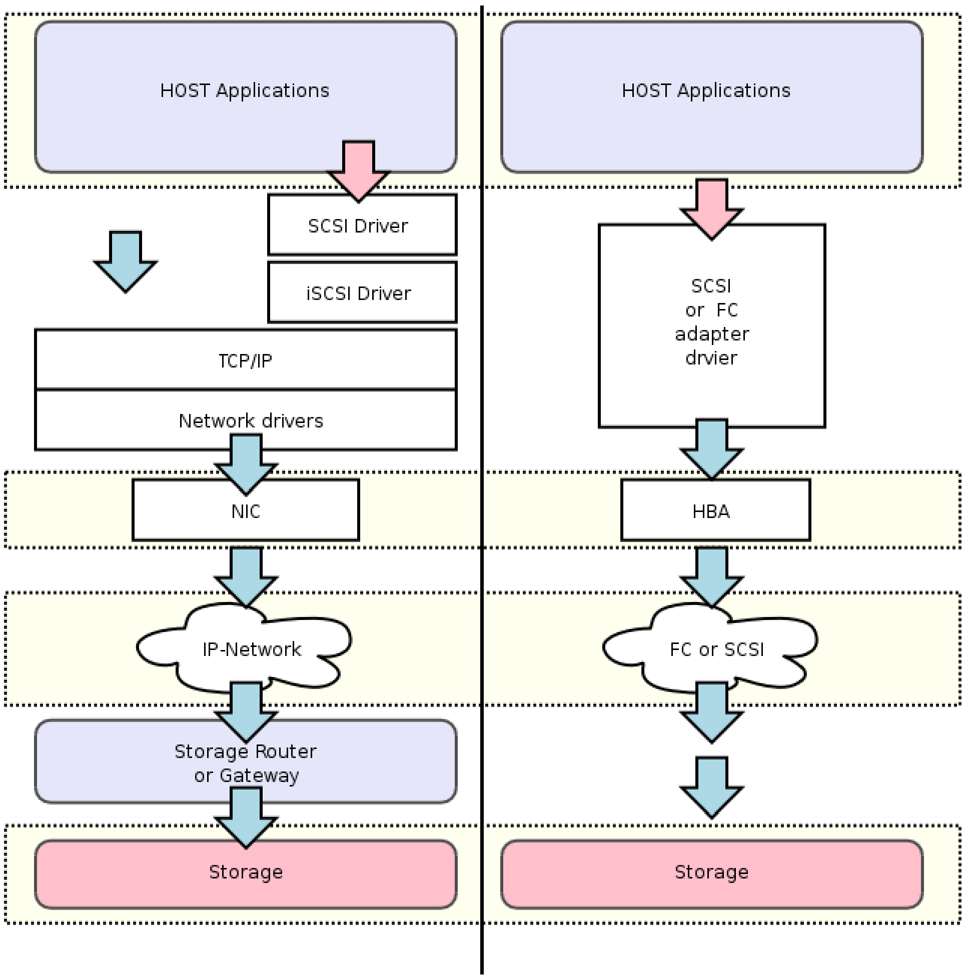

Figure 5.1: iSCSI compared to DAS and FC SAN.

existing Ethernet infrastructures can be used. Storage device manufacturers also believe that existing devices could be more easily adapted, if the command sets remains close to the already well known SCSI command set.

Back in the 80ties only few people believed that 10MBit Ethernet could be enhanced. Now the first 10GBit network devices hit the store. Ethernet has matured over time from a shared media network with low reliability, to a very high performance switched networking system. And iSCSI builds on these enhancements in Ethernet technology to achieve its performance goals. 10GBit Ethernet and NICs with TCP Offload Engines (TOE) will be an interesting opportunity for iSCSI based SANs.

5.2 NAS over TCP/IP - NFS and NFSv4

NFS has a long history as a popular Unix network filesystem. NFS is a typical application of a Remote Procedure Call (RPC) architecture. All file system operations are forwarded via RPC to another host. The procedure arguments and return values are marshalled in a special format, called External Data Representation (XDR).

NFS has the following characteristics:

- Design for easy recovery.
- Independent of transport protocols and operating systems.
- Simplicity.
- Good Performance.

The functionality of NFS is defined in terms of RPC messages. Up to version 4, each procedure represents a particular action that a client may perform, like reading from a file, writing to a file, creating or removing a directory. For the server to perform any action, you have to pass the server a valid file handle. It is much like performing local file system operations with a longer wire.

5.2.1 NFS Version 2 and 3

NFS exists in 3 major flavors: Version 2 and 3, and the only recently released Version 4. NFS Version 2 and 3 are pretty similar. In NFSv2 and v3 all actions map directly into RPC procedures. Initially RPC was only done via the unreliable UDP. Starting with NFSv3, TCP could also be used.

Up to version 3, NFS was a stateless protocol. That means the server doesn't keep any information about the clients between requests. No state is lost if the server crashes. File handles, the central concept of file manipulations, must thus be unique. NFS directly exports Unix inode numbers as file handles, which uniquely identify files on a local filesystem. This was a very efficient way of performing remote file operations. But this also led to problems when using NFS on non Unix platforms, which often don't employ the technique of inodes.

NFSv2 and v3 really are a whole suite of protocols on top of RPC. The suite consisted of:

- mount protocol (MNT)

- network file system (NFS)
- network lock monitor (NLM)
- network status monitor protocol (NSM)

The mount protocol provides operating system specific services to get NFS off the ground. It looks up server path names, validates user identities, and checks access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote file system.

The mount protocol was kept separate from the NFS protocol to make it easy to plug in new access checking and validation models without changing the NFS protocol. The only information shared between the mount protocol and NFS up to version 4, is the file handle structure. In a typical Unix NFSv3 implementation this daemon is called `mountd`.

The network status monitor protocol (NSM) is needed for the lock monitor, which needs per host status information, such as reboots. This protocol is often provided by a daemon called `statd`.

Locking is a stateful protocol. In Version 2 and 3 of NFS, locking was not mandatory. In order to keep the NFS protocol slim and simple, locking is the job of a separate service. The network lock monitor knows which files are in use and manages lock information. It works in conjunction with the NSM protocol in order to release locks after a client reboot has happened. This service is often implemented as `lockd`.

Remember that the whole NFS suite works over RPC. So in addition to these protocols, a typical NFSv3 implementation needs a SUN RPC stack, and a special service called `portmap` or `rpc.portmap`. It is a so called portmapper, that maps RPC program numbers to TCP or UDP port numbers. Before sending a request to a RPC service, a client has to talk with a portmapper to get the port number for that service.

5.2.2 NFS Version 4

NFS Version 4 (NFSv4) is a major change in the NFS design. The requirements for NFSv4 were:

- Improved access and good performance on the Internet.
- Strong security, with security negotiation built into the protocol.
- Enhanced cross-platform interoperability.

- Extensibility of the protocol.

Among the most important changes are:

- Eliminating helper protocols (nfs, mountd, nlm, nsm).
- Introduction of COMPOUND to reduce roundtrip time.
- Statefulness (introduction of Open and Close).
- Works better across firewalls.
- Adds strong security features.

5.2.3 NFS example

A simple operation like reading a file named “x/work.txt” involves the following abstract file system operations:

- Mount the remote filesystem
- Open file handle
- Read file (multiple times)
- Close file

To mount a filesystem means making it accessible through the UNIX filesystem API. A Windows(TM) user can think of mounting as mapping a network drive, only that you can mount a file system anywhere in the file tree.

In a typical unix environment, one would write the following commands to mount the subfilesystem `vol0` from `zeus` to the local directory called `mnt` and read the first of 32 kBytes from that remotely mounted file system.

```
mount zeus:/export/vol0 /mnt
dd if=/mnt/home/data bs=32k count=1 of=/dev/null
```

Using NFSv3, according to [17] the following sequences result from such an operation:

```
  -> PORTMAP C GETPORT (MOUNT)
  <- PORTMAP R GETPORT
  -> MOUNT C Null
  <- MOUNT R Null
  -> MOUNT C Mount /export/vol0
  <- MOUNT R Mount OK
  -> PORTMAP C GETPORT (NFS)
  <- PORTMAP R GETPORT port=2049
  -> NULL
  <- NULL
  -> FSINFO FileHandle=0222
  <- FSINFO OK
  -> GETATTR FileHandle=0222
  <- GETATTR OK
  -> LOOKUP FileHanlde=0222 home
  <- LOOKUP OK FileHandle=ED4B
  -> LOOKUP FileHandle=ED4B data
  <- LOOKUP OK FileHandle=0223
  -> ACCESS FileHandle=0223 (read)
  <- ACCESS OK (read)
  -> READ FH=0223 at 0 for 32768
  -> READ OK (32768 bytes)
```

This above sequence contains simplified output from an actual network trace. Each of the 11 request and response pairs represent a network roundtrip.

The following traffic would result in an NFS version 4 network:

```
=> PUTROOTFH
     LOOKUP ‘‘export/vol0’’
     GETFH
     GETATTR
  <= PUTROOTFH OK CURFileHandle
     LOOKUP OK CURFileHandle
     GETFH OK
     GETATTR OK
  => PUTFH
     OPEN ‘‘home/data’’
     READ at 0 for 32768
  <= PUTFH OK CURFileHandle
     OPEN OK CURFileHandle
     READ OK (32868 bytes)
```

In the above example, the number of round trip requests for the same application in NFSv4 compared to prior versions is reduced from 11 to two request and response transactions.

5.3 Performance of SAN and NAS protocols

There are two big families of distributed storage systems - SAN and NAS architectures. These two vary mainly in the way network storage is seen. SAN remains a block based view of the data, which closely resembles the way DAS works. This leads to more flexibility and better throughput, because of fewer overhead. NAS on the other hand exports file storage to its clients. Operations are done on a file level, hiding the complexity of raw storage inside the NAS server.

In NAS systems not only the storage is outsourced from the local host, but also the whole file system, leading to slimmer host computers. This reduces the TCO of server systems. But the NAS, beeing a high level file server, adds much overhead to the storage device (NAS server). Such a NAS server can soon become a bottleneck, especially in IO heavy applications with multiple concurrent accesses.

In a recent paper two IP-networked storage protocols, one SAN and one NAS protocol, are compared [18]. As already mentioned SAN protocols use a block based access, whereas NAS protocols provide file based access to data. As in the mentioned paper, these two will be referred to as *block-access* and *file-access* protocols. The block-access protocol is the already introduced iSCSI protocol, and the file-access protocol is NFSv4.

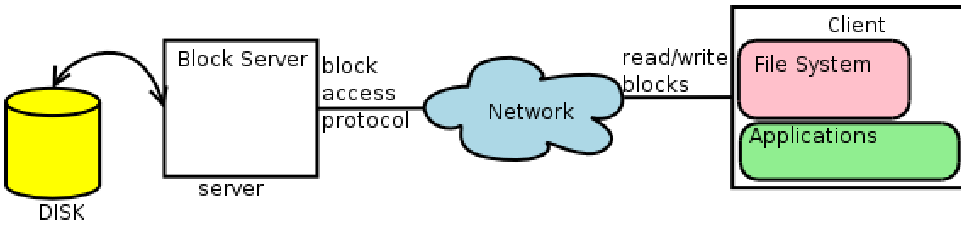

Figure 5.2: Block access protocol.

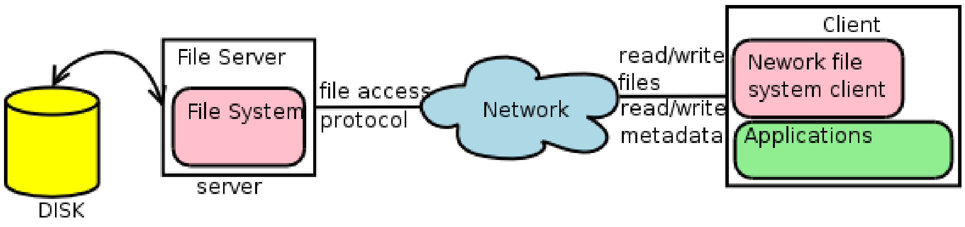

Figure 5.3: File access protocol.

A very important performance related difference of the two approaches is that, although both protocols persist the metadata on the server, in the block-access protocol (see figure 5.2) the file system resides in the clients memory and only gets serialized to special blocks on the server. In file-access protocols (see figure 5.3) the meta-data resides entirely in the memory of the server - if the client wants to do meta data access, it has to issue network operations, for instance Remote Procedure Calls (RPC).

Compared to the block-access protocols, the server based metadata storage in file-access protocols has a key asset in that sharing of storage between multiple clients is much easier. There is one node, the server, that manages all the metadata. In this comparison a point-to-point infrastructure is assumed - one client talks to one server.

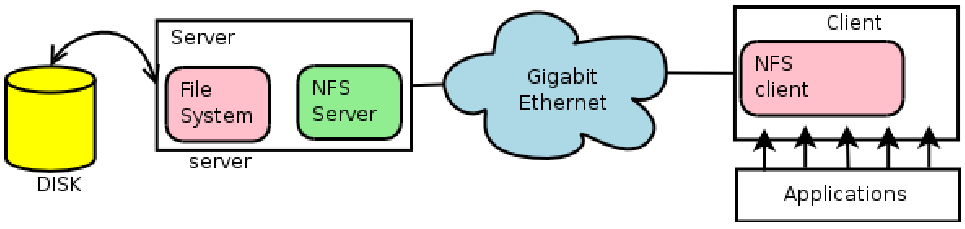

Figure 5.4: Typical NFS NAS Setup.

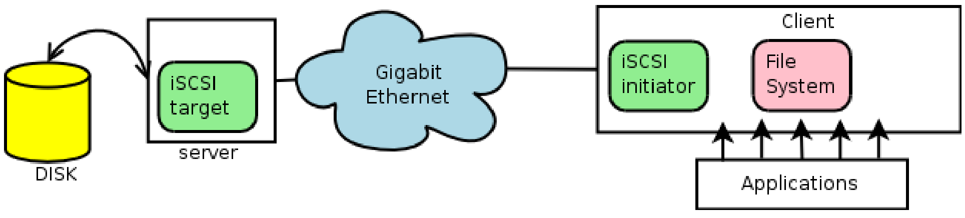

Figure 5.5: Typical iSCSI SAN Setup.

[18] concludes that the performance of NFS and iSCSI is comparable for data intensive workloads, while iSCSI outperforms NFS by a factor of 2 or more for meta-data intensive workloads. Aggressive meta-data caching and update aggregation of iSCSI is the main reason for this performance difference. By enhancing NFS to improve its meta-data performance, the authors show that this bottleneck of NFS can be reduced.

5.4 Where are we today?

A decade ago almost nobody believed in the high scalability of Ethernet. But Gigabit Ethernet, intelligent Ethernet switches and high performance network hardware along with very stable IP stacks have proven that existing network infrastructure is an interesting option for storage networks. Even big server systems like Google, CERN, and many more, are employing NAS technology today. The advantage of NAS is that existing storage technologies can be reused and accessed on a file basis over the network. But the question of the unified network file system is by no way clear. Small installations would use popular network file systems like CIFS or NFS, but big companies are building on their own implementations, as shown in [4].

One of the biggest drawbacks of SAN technology is its high price and its rare employment. Also many of the disadvantages of cheap storage hardware has changed as S-ATA has proved itself as a very high performance technology. Even [**?**, TheCERNDiskStorage0405]refers S-ATA over SCSI/FC devices for high availability.

But SAN vs. NAS is not always an either or. Both technologies can also be used together. SAN is positioned on a much lower level and can be used to implement a high performance NAS system. On the other hand the use of iSCSI in conjunction with Gigabit Ethernet or 10 Gigabit Ethernet is very promising. According to iSCSI, it is not ready for prime time yet. The advantage of iSCSI is that it can be used to build storage servers based on conventional DAS storage. For instance a big storage server based on S-ATA can function as an iSCSI target on a high speed network, thus pushing SAN technology into mid-level server installations.

List of Figures

Bibliography

[1] Tony Cass. CERN disk storage technology choices, 2005.

[2] Ming-Syan Chen, Hui-I Hsiao, Chung-Sheng Li, and Philip S. Yu. Using rotational mirrored declustering for replica placement in a disk-array-based video server. In *ACM Multimedia*, pages 121–130, 1995.

[3] Garth A. Gibson, William V. Courtright II, Mark Holland, and Jim Zelenka. RAIDframe: Rapid prototyping for disk arrays. Technical Report CMU-CS-95-200, 1995.

[4] Sanjay Ghemawat Howard Gobioff and Shun-Tak Leung. The google file system, 2003.

[5] Xubin He, Ming Zhang, and Qing Yang. Stics: Scsi-to-ip cache for storage area networks. *J. Parallel Distrib. Comput.*, 64(9):1069–1085, 2004.

[6] J. L. Hennesy and D. A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 1990.

[7] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Journal of Distributed and Parallel Databases*, 2(3):295–335, 1994.

[8] W. Holland, G. Gibson, L. Reilly, and J. Zelenka. Raidframe: A rapid prototyping tool for raid systems, 1996.

[9] Kai Hwang, Hai Jin, and Roy Ho. RAID-x: A new distributed disk array for I/O-centric cluster computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 279–287, Pittsburgh, PA, August 2000. IEEE Computer Society Press.

[10] Kai Hwang, Hai Jin, and Roy S.C. Ho. Orthogonal striping and mirroring in distributed raid for i/o-centric cluster computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):26–44, 2002.

[11] W. Courtright II. *A Transactional Approach to Redundant Disk Array Implementation.* PhD thesis, Carnegie Mellon, April 1997.

[12] W. Courtright II, G. Gibson, M. Holland, and J. Zelenka. A structured approach to redundant disk array implementation, 1996.

[13] W. V. Courtright II, G.A. Gibson, M. Holland, and J. Zelenka. Raidframe: Rapid prototyping for disk arrays. In *Proceedings of the 1996 Conference on Measurement and Modeling of Computer Systems*, pages 268–269, May 1996.

[14] E. Lee and R. Katz. Performance consequences of parity placement in disk arrays. pages 190–199. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[15] D. D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. SWIFT/RAID: A distributed RAID system. Technical Report UCSC-CRL-94-06, 1994.

[16] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. Ibm storage tank– a heterogeneous scalable san file system. *IBM Syst. J.*, 42(2):250–267, 2003.

[17] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[18] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy. A performance comparison of nfs and iscsi for ip-networked storage, 2004.

[19] Philipp Reisner. Drbd - distributed replicated block device. August 2002.

[20] Philipp Reisner. Rapid resyncronization for replicated storage. January 2004.

[21] Raj Srinivasan. XDR: External Data Representation Standard. Technical Report 1832, 1995.