

# Distributed Storage Exam Notes

---

## Contents

01: Socket Programming	2
02: Remote Procedure Calls & Network File System	4
03: Andrew File System & Reliable Storage: Single Server	8
04: RAID systems	13
05: Reliable Storage: Finite Fields & Linear Coding	23
06: Repair Problem & Regenerating Codes	30
07: Reliable Storage: Regenerating Codes & Local Repairability	36
08: Hadoop Distributed File System (HDFS)	40
09: Storage Virtualisation	50
10: Object Storage	58
11: Compression & Delta Encoding	59
12: Data Deduplication	60
13: Fog/Edge Storage	61
14: Security in Storage Systems	62

# 01: Socket Programming

## Data Storage

---

### Basic Problem

- ◆ Persistent storage
- ◆ Reliable storage: no corruption, minimise loss
- ◆ Cope with large amounts of data

### Discs

- ◆ Exponential growth in capacity
- ◆ Linear growth in read/write speeds

Thus simply getting larger discs might not help, and will inversely effectively slow down the overall system - especially for HDDs where random access is slow, and having to get more data from a single drive simply overburdens it.

### Distributed System

- ◆ Single machine won't be enough
- ◆ Interface with multiple machine
  - ◆ Coordination (Interesting challenge)
  - ◆ Programming is more complex
  - ◆ Synchronise and manage
  - ◆ Network limitations
- ◆ More points of failure
- ◆ Limitation of components
- ◆ Expect continuous failure

NFS bad quote: "*still in use, sometimes I wonder why*" ☒

### ☒ Possible Report Weakness

Was I supposed to do replication with erasure codes??

### HDFS

- ◆ Get data quickly
- ◆ Analyse quickly

### ☒ Possible Report Weakness

Not expected to do low level socket programming, expected to use ZMQ or Flask or such.

### ☒ Possible Report Weakness

Definitely supposed to have done parallelism

**1. Main thread to accept connections**

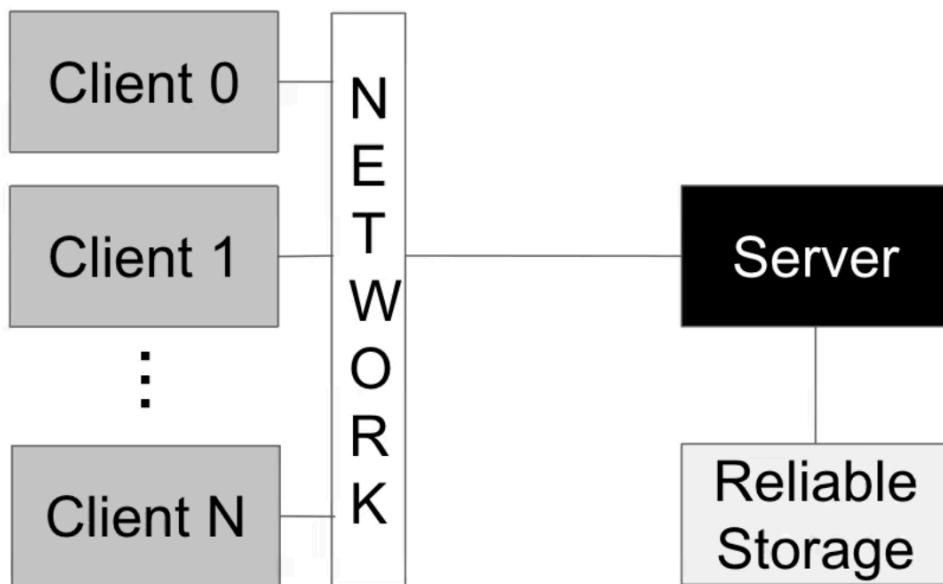
→ Make a new socket to handle each connection on a different socket.

**2. Event-based reaction**

→ Decent, but different

Wouldn't do this in Python

## 02: Remote Procedure Calls & Network File System



Abstract away the remote file system as if it were local

### Remote Procedure Calls

---

- ◆ NFS is defined as a set of RPCs
- ◆ Almost 1-to-1 match with local file system commands
- ◆ RPCs are synchronous

### Performance

- ◆ Limited by RPC overhead
  - ◆ Small packets → overhead dominates
  - ◆ Large packets → transmission time dominates

### Characteristics

- ◆ Nearly stateless
  - ◆ Need to retry if fail △
- ◆ **Things that are not stateless:**
  - ◆ Keeps some RPC reply cache
  - ◆ Certain file locking
  - ◆ *File handle* - reference to the file that an action is to be applied on

### Network File System

---

- ◆ *Continuous changes* over network
  - Opposite from AFS, which only applies changes once the user finishes a modification
- ◆ OS *independence*
- ◆ Simple *crash recovery* for clients and servers
- ◆ *Transparent* access (abstract away that anything is remote)
- ◆ Maintain *local file system semantics*
- ◆ Reasonable *performance*

### File Handle *fhandle*

- ◆ **file handle:** reference to file to act on
- ◆ **offset:** where to start in the file
- ◆ **count:** how many bytes to read

## NFS Protocol - File Operations (not complete)

### NFSPROC\_GETATTR

expects: file handle  
returns: attributes

**Return file attributes**

### NFSPROC\_SETATTR

expects: file handle, attributes  
returns: nothing

**Set file attributes**

### NFSPROC\_LOOKUP

expects: directory file handle, name of file/directory t  
returns: file handle

**Look file in directory**

### NFSPROC\_READ

expects: file handle, offset, count  
returns: data, attributes

**Reads up to count bytes with offset bytes from file**

### NFSPROC\_WRITE

expects: file handle, offset, count, data  
returns: attributes

**Writes count bytes with offset bytes to file**

### NFSPROC\_CREATE

expects: directory file handle, name of file, attributes  
returns: nothing

**Creates new file name**

### NFSPROC\_REMOVE

expects: directory file handle, name of file to be removed  
returns: nothing

**Removes file name**

### NFSPROC\_MKDIR

expects: directory file handle, name of directory  
returns: file handle

**Create new directory name in a directory**

### NFSPROC\_RMDIR

expects: directory file handle, name of directory  
returns: nothing

**Remove directory**

### NFSPROC\_READDIR

expects: directory handle, count of bytes to read  
returns: directory entries, cookie (to get more)

**Returns up to count bytes of directory entries from the directory.**

**A cookie is used in subsequent readdir calls to start reading at a specific entry in the directory.**

**A cookie of zero returns entries starting with the first entry in the directory**

**NFSPROC\_NULL**  
expects: nothing  
returns: nothing

Used for pinging the server

**NFSPROC\_STATFS**  
expects: file handle  
returns: file system stats

Returns file system stats:  
block size, number of free  
blocks, ...

More than the above.

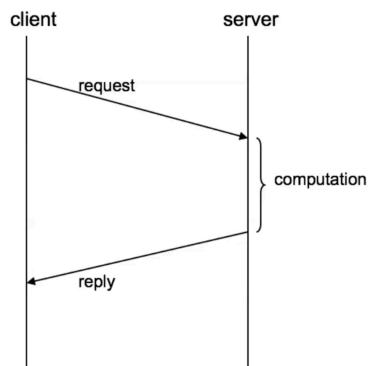
- ◆ **MOUNT:** protocol takes a directory pathname and returns an fhandle
- ◆ UNIX-style permission checks, where this information goes through the RPC
- ◆ Root-access on client gives root access on server? NO smh ☹

### Idempotency

Property of certain operations where even if applied multiple times, the end result will be the same, as if it were *only ever applied once*.

- ◆ This means you can always *retry* without any issues.
- ◆ **mkdir:** not idempotent

### Request-Reply



### Client-side Caching

- ◆ Caching and write buffering to improve performance
  - ◆ **Why:** Sending command at each *ctrl+s* from the user is extreme protocol overhead. The *network* speed shouldn't be relied on and taken for granted in that way.
  - ◆ **What:**
    - *1 user:* Temporary local buffer is good, but this fails with 2 users modifying the same file, or just one of them looking at it, the changes won't be applied.
    - Once buffer is closed → client flushes data → other users can see it.
  - ◆ **Problem; stale cache** → this can be mitigated somewhat by sending a GETATTR to see when the file has been modified last.

### NFS Versions

Later versions build upon **NFSv2**

**NFSv4.1** first version to have *multiple servers*

### ☒ Possible Report Weakness

Should have used some RPC implementation. `tinyrpc`?

I suppose I did this somewhat by sending JSON messages with message type and thus the *storage nodes* were able to know how to handle and respond to each message.

# 03: Andrew File System & Reliable Storage: Single Server

## Andrew File System

- Does have a way to create multiple replicas, and can decide where to place them  
→ Mimics something like [NFSv4.1](#)

**Server:** Vice

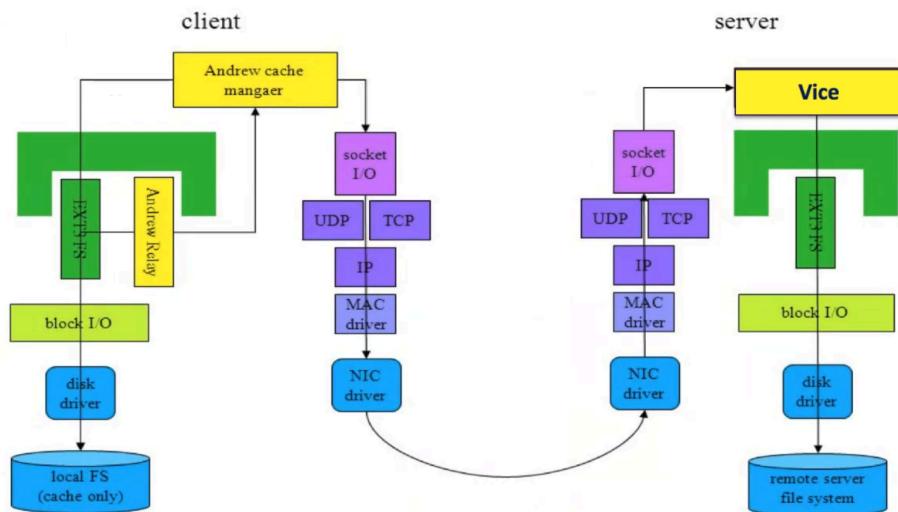
**Client:** Venus

### Main Goal

### Scalability

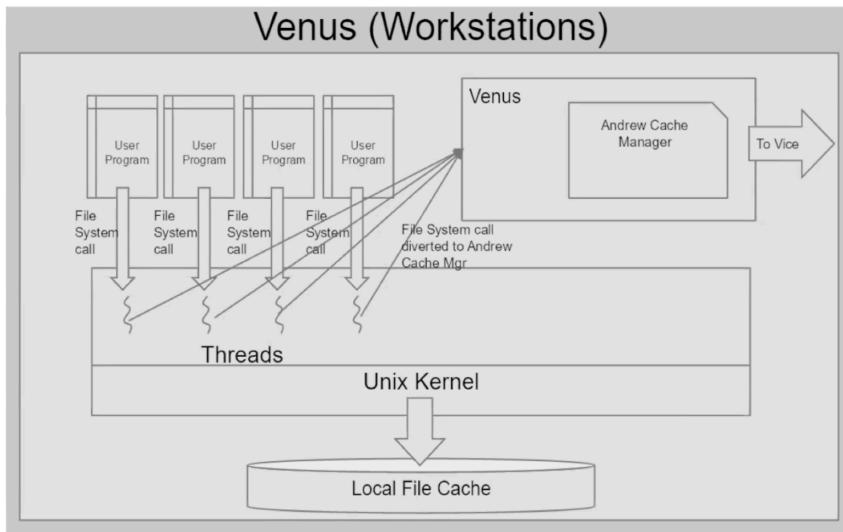
### Properties

- Scalable
- Whole-file* caching on local disk of client
  - Allow for many modifications at once, only sending large protocol messages
  - Engages the local file system
  - Less messages → less protocol overhead



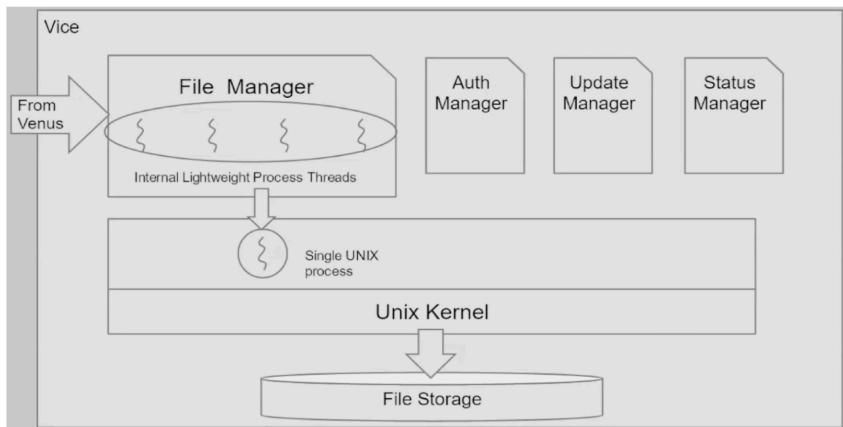
*Note that the local file system has to exist and be engaged.*

### Venus (Client)



- ◆ Decisions on caching
  - ◆ what to do when cache space has run out.
- ◆ Multiple threads

## Vice (Server)



- ◆ Single thread
  - ◆ Serving data
  - ◆ Storing data

## Organising Data

### Local volumes:

- ◆ Logical unit corresponding to a directory
- ◆ *Difference to NFS*: Allows operators to deal with logical volumes; replicating, moving, otherwise modifying
- ◆ User doesn't need to be aware where the logical volumes point

### Benefits:

- ◆ Load-balancing → increased availability & performance
- ◆ Can be moved easily

## Callback

### Goal:

- ◆ Ensure cached copies of files are up-to-date when another client closes the same file after updating it.

### Callback promise:

- ◆ Token issued by Vice
- ◆ Token stored with the cached files on local client disk
- ◆ Token is valid or cancelled
  - **Valid:** Free to modify
  - **Cancelled:** Do not touch  $\Delta$

### Implementation:

- ◆ RPC from Vice to Venus
- ◆ Set token to cancelled

## Consistency Guarantees

Two clients open one file and makes changes. Server will only be updated on close. What happens here?

- Clients need to know when the Vice is updated to know that their copy is *old*
- Upon close, *promises* to show updated data to all clients

- ◆ Weak
- ◆ Practical
- ◆ Though with some pitfalls

### Methods to provide this guarantee:

- ◆ **Write-through Cache:** Once modified on client, it will be modified on server (on close)
- ◆ **Callbacks:** Setting other tokens to cancelled, making other clients aware that they have to re-fetch.

## THE ANDREW FILE SYSTEM (AFS)

P <sub>1</sub>	Client <sub>1</sub> P <sub>2</sub>	Cache	P <sub>3</sub>	Client <sub>2</sub>	Cache	Server Disk	Comments
open(F)	-	-	-	-	-	-	File created
write(A)	A	-	-	-	-	-	
close()	A	-	-	-	-	A	
	open(F)	A	-	-	-	A	
	read() → A	A	-	-	-	A	
	close()	A	-	-	-	A	
	open(F)	A	-	-	-	A	
	write(B)	B	-	-	-	A	
	open(F)	B	-	-	-	A	
	read() → B	B	-	-	-	A	
	close()	B	-	-	-	A	
		B	open(F)	A	A	A	Local processes see writes immediately
		B	read() → A	A	A	A	
		B	close()	A	A	A	
	close()	B	A	-	-	B	Remote processes do not see writes...
		B	open(F)	B	B	B	
		B	read() → B	B	B	B	
		B	close()	B	B	B	
		B	open(F)	B	B	B	
	open(F)	B	B	B	B	B	
	write(D)	D	B	B	B	B	
		D	write(C)	C	B	B	
		D	close()	C	C	C	
	close()	D	A	-	-	D	... until close() has taken place
		D	open(F)	D	D	D	
		D	read() → D	D	D	D	
		D	close()	D	D	D	
							Unfortunately for P <sub>3</sub> the last writer wins

## AFSv2

- ◆ Able to support 50 clients per server
- ◆ Client-side performance close to local performance - since everything basically happens in local cache

## AFS vs NFS

# AFS vs NFS

	AFS	NFS
Supported Clients	Requires Disks	Diskless (client side)
Network Traffic	As required (much less)	Periodic (high)
Server Load	As required (much less)	Periodic (high)
Client Response Times	Equal	Equal
OS Support	Minimal	More
Caching Protocol	Whole Files	Blocks of Files
Caching Mechanism	Client Disks	Client Memory
Security	Kerberos	Weaker more primitive
Admin Management	Simpler	Difficult
Server	Stateful	Stateless

## Reliable Storage: Single Server

---

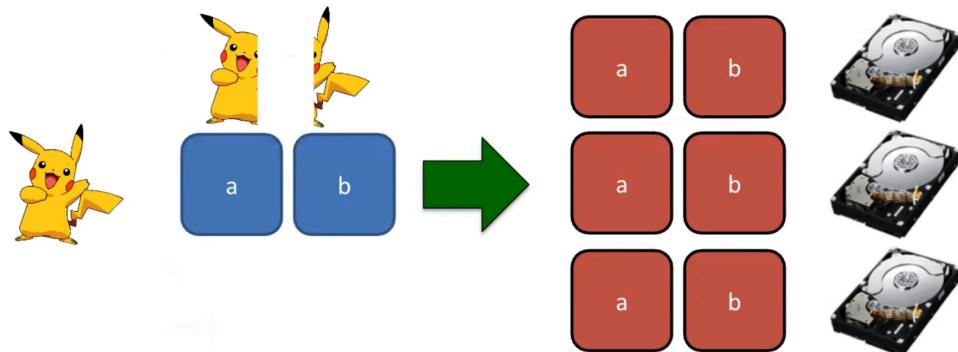
### Basic Problem

- ◆ No corruption
- ◆ Minimise loss

Individual servers are unreliable, some redundancy will be needed.

Start thinking *one machine* with multiple disks

### State of The Art: Replication



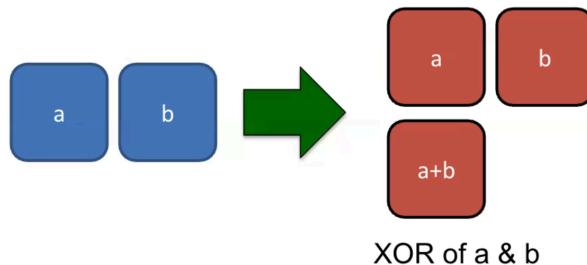
**Overhead:** 200%

- ◆ Make decisions about where to place each chunk/fragment/stripe/slice; impacts reliability

- ◆ How many slices to make:
  - defines how much is lost if one fragment is impacted
  - defines how big of an action is needed to recover, like moving 10/20/50 percent of total file.

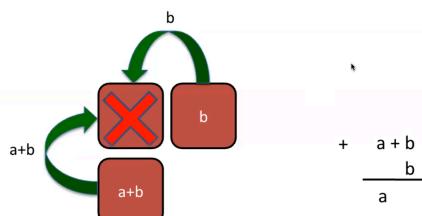
**Traffic to repair:** 1 unit, 50% of file size

### Different Options: Vanilla RAID5



**Overhead:** 50%

- ◆ Can lose *any one of the shown fragments*, and still recover it.
- ◆ Need to move *the equivalent* of the entire file, no matter how many slices.



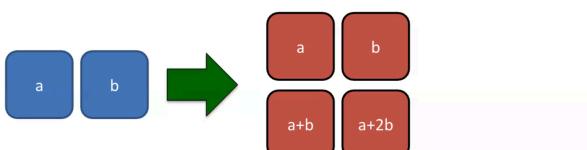
Overhead: 50%

Protection: 1 loss

Traffic to repair loss: 2 units (100% of file size)

### Different Options: Vanilla RAID6

- ◆ Sustain *any two losses*, while only storing twice the data
- ◆ Slicing thinner gives better performance, with further linear combinations of the slices.



Overhead: 100%

Protection: 2 losses

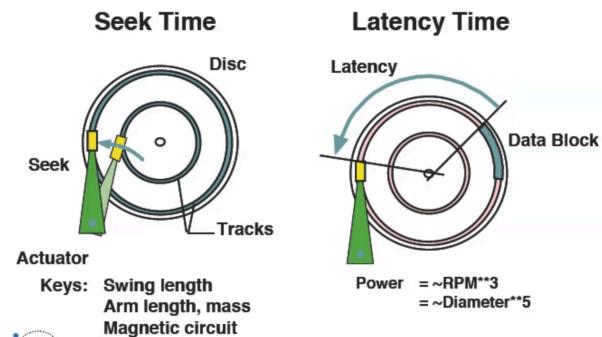
Traffic to repair loss: 2 units (100% of file size)

# 04: RAID systems

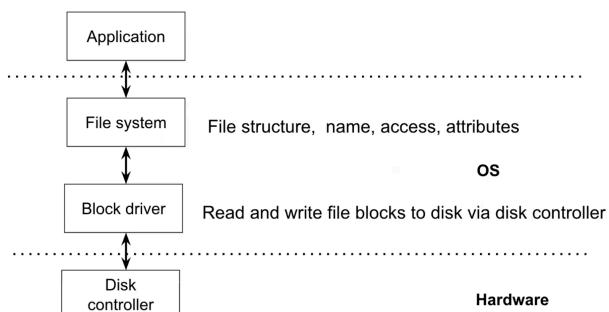
## Properties

- Reliability
- Increase *speed* beyond individual disk capabilities
  - Since disk speed growth has plateaued

## Disk



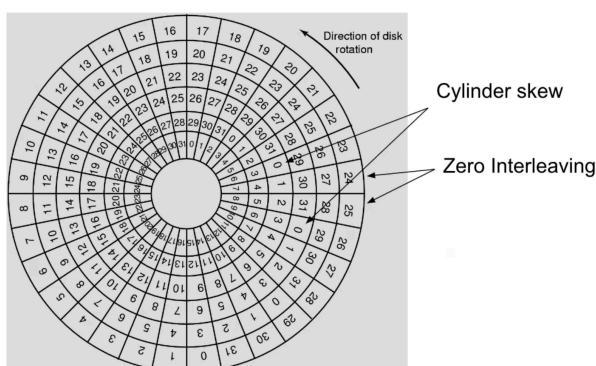
$$\text{access time} = \text{seek time} + \text{latency time} + \text{transfer time} \quad (1)$$



## Disk Arm Scheduling Algorithm

- Seek time - Move to the right cylinder
- Latency time - Sectors rotate under the head
- Transfer time - Moving the data to/from memory

**Dominant:** physical seek time and physical rotation



### FIFO:

- ◆ Sub-optimal
- ◆ *Very fair*

### SSTF: Shortest Service Time First

- ◆ Might never finish a specific request
- ◆ Or big delays for some requests
- ◆ *Not fair*

### SCAN:

- ◆ Always moving arm side to side
- ◆ Read whichever request is there no matter when it was given
- ◆ Favoring outermost and innermost tracks

### C-SCAN:

- ◆ Best ☺

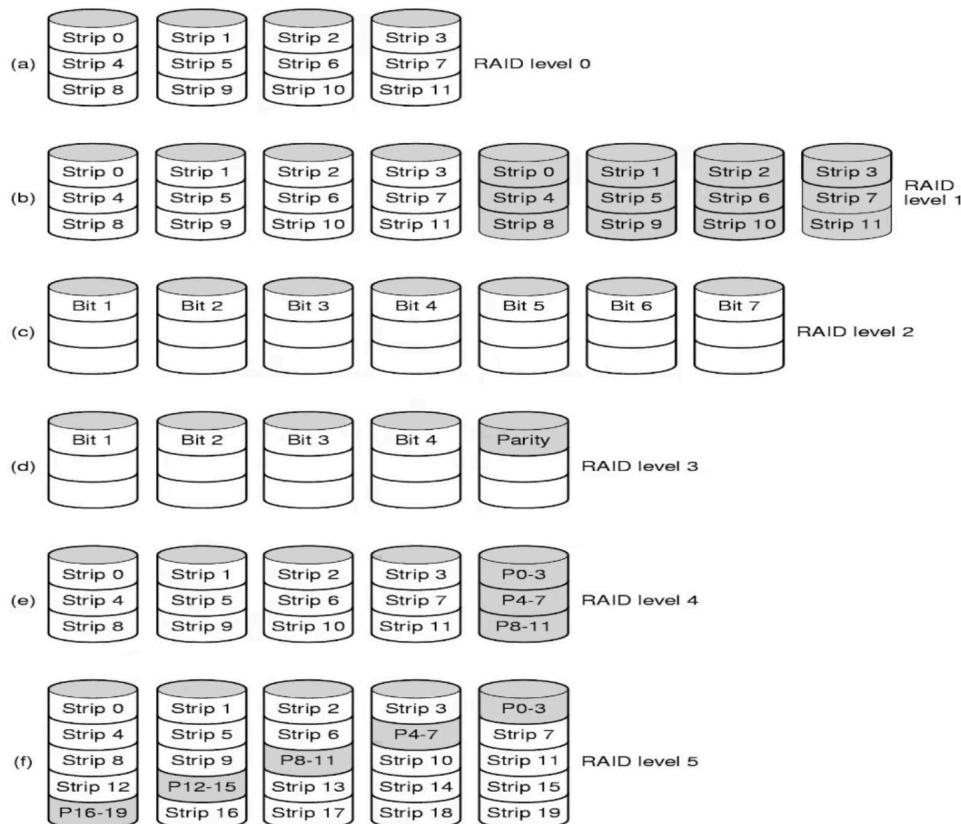
## RAID

### Redundant Array of Inexpensive Disks

#### ☒ Possible Report Weakness

Didn't use any RAID terminology in report; *make sure to be able to use it at the exam*

What is possible with many inexpensive disks?



## Metrics

- ◆ **Space Efficiency:** File size divided by the space used in the system.
- ◆ **Storage Capacity:** Amount of data that can be stored in the system given a number of devices.
- ◆ **Fault Tolerance:** Number of failed devices,  $F$ , that results in loss of data.
- ◆ **Read Performance:** Given a system read speed of  $R_{r,s}$  and a read speed of an individual disk  $R_{r,d}$ , the read performance is the speed-up:  $R_{r,s}/R_{r,d}$ .
- ◆ **Write Performance:** Same as read performance, but with write speed of the system  $R_{w,s}$  and of the individual disk  $R_{w,d}$ :  $R_{w,s}/R_{w,d}$
- ◆ **Minimum Number of Disks Required**

*Note: difficult to work with these metrics if all devices/disks aren't exactly the same.*

## ☒ Possible Report Weakness

Make sure to be able to use the *above metrics* talking about the report.

## What is RAID

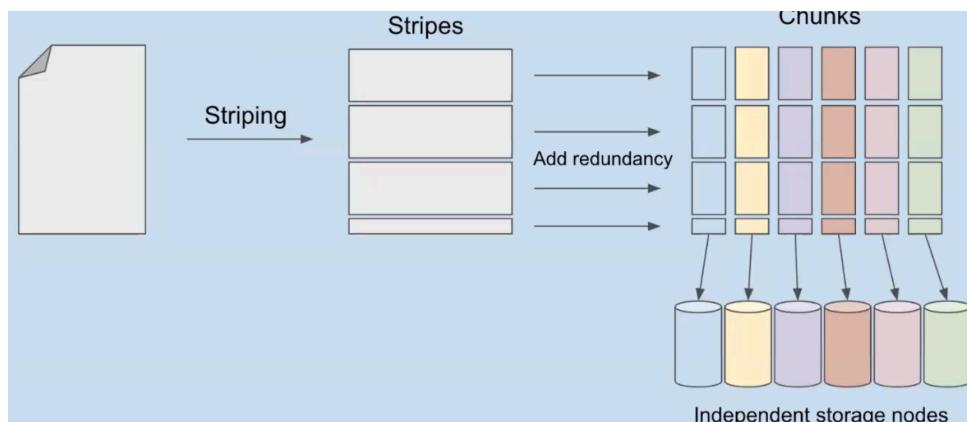
**Redundant Array of Inexpensive Disks**

**Redundant Array of Independent Disks**

Alternative to SLED: Single Large Expensive Disk

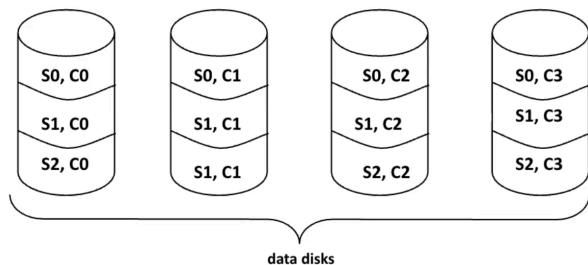
- ◆ A RAID box with a RAID controller looks just like a SLED to the system using it

## Striping



## RAID0

- ◆ Non-redundant
- ◆ *Striped* and *chunked*
- ◆ Not any more space efficient



$Sx, Cy = \text{Stripe } x, \text{ Chunk } y$

### Purpose:

- ◆ Increase speed
- ◆ Read/write in parallel from multiple devices
- ◆ Serve data very quickly

### Disk failure:

- ◆ Results in many stripes affected
- ◆ Can't recover at all
- ◆ Reliability is worse than SLED

## RAID0

Metric	Evaluation
Space Efficiency	1
Storage Capacity	$N \times C$ , For $N$ equal sized disks with capacity $C$
Fault Tolerance	$F = 1$ , a single disk causes damage
Read Performance	$N$
Write Performance	$N$
Min Disks Required	2

### RAID1

- ◆ Meant for *reliability*
- ◆ **Key:**  $N$  mirrored disks
  - ◆ 1 Main disk,  $N - 1$  mirrors
  - ◆ Full file replication
- ◆ **Failure:** Use any surviving disk, plug in new disk, copy over
- ◆ **Read:** Pick fastest disk,  $\sim 2x$  for  $N = 2$ , with a lot of requests, parallel read from several drives for different files.
- ◆ **Write:** Single disk speed, no speed-up

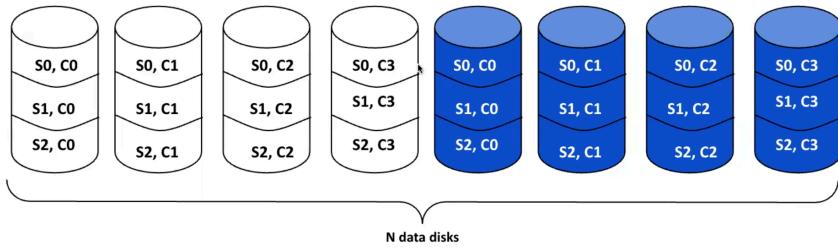
## RAID1

Metric	Evaluation
Space Efficiency	$1/N$ , very <i>bad</i>
Storage Capacity	Capacity of <i>smallest</i> single disk
Fault Tolerance	$F = N$ , lose all disks
Read Performance	$N$ , only in high load scenarios
Write Performance	1, need to write to each disk
Min Disks Required	2

### RAID1 E

- ◆ Merge RAID1 with ideas of striping & chunking from RAID0
  - ◆ *Striped* and *chunked*
- ◆ **Read:** Can approach that of RAID0
- ◆ Typically not interleaved

*Basically RAID0 with one replica of everything*



$S_x, C_y = \text{Stripe } x, \text{ Chunk } y$

### RAID1 E

Metric	Evaluation
<b>Space Efficiency</b>	$1/2$ , typically used with 2 replicas
<b>Storage Capacity</b>	$NC/2$ , half capacity of $N$ disks
<b>Fault Tolerance</b>	$F = 2$ , can cause loss in worst case, best case: $F = N/2$
<b>Read Performance</b>	$N/2$ , light load, up to $N$ under high load
<b>Write Performance</b>	$N/2$ (could be higher depending on placement strategy)
<b>Min Disks Required</b>	3

### ☒ Possible Report Weakness

The idea that read/write could be higher depending on placement strategy is introduced here the first time, and is likely something like that that would have been nice to see in the project and report.

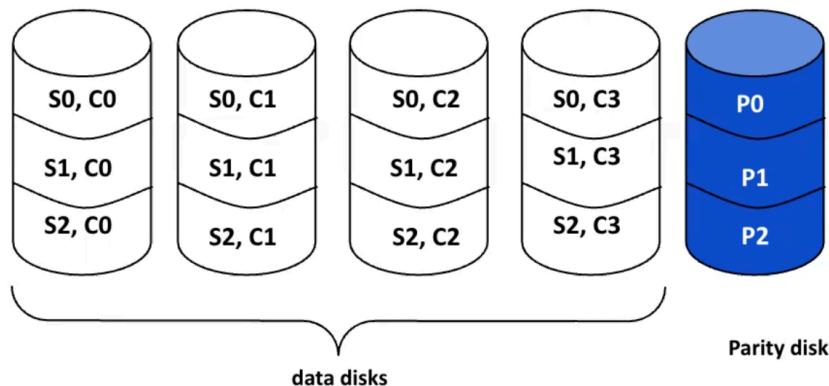
However, it does of course require some parallel setup like this, which I didn't have.

*Refer to this (RAID1 E) if asked about that*

### RAID4

- ◆ {Block/stripe}-level parity with stripes
- ◆ **Read:** accesses all the disks
- ◆ **Write:** accesses all the disks **and** the parity disk
- ◆ *Heavy load on parity disk*

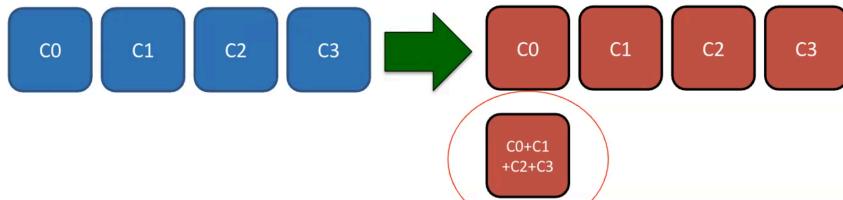
Like RAID0 but with parity



$S_x, C_y, P_z = \text{Stripe } x, \text{ Chunk } y, \text{ Parity } z$

### Parity:

- For each stripe create a parity
- bit-by-bit xor for between chunks for each stripe




---

### RAID4

Metric	Evaluation
Space Efficiency	$1 - 1/N$ , taking the parity disk into account, otherwise like RAID0
Storage Capacity	$(N - 1) \times C$ , for $N$ disks with the same capacity of $C$
Fault Tolerance	$F = 2$ , of the data disks, as at least two are needed to recover the combination of them
Read Performance	$N - 1$ , taking the parity disk into account, otherwise like RAID0
Write Performance	$N - 1$ , taking the parity disk into account, otherwise like RAID0
Min Disks Required	3, 2 disks for striping and chunking, 1 for parity

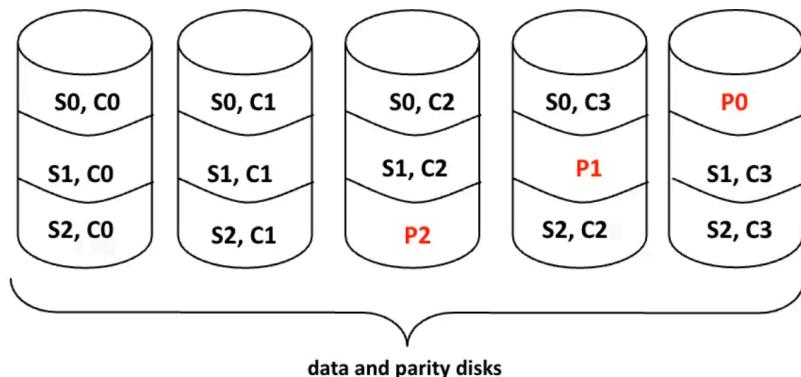
### Challenges

- Separated parity disk to one single disk
- Replaced by RAID5

### RAID5

- Block-interleaved parity
- Can activate more disks when needing parity
- Read:** Better read than RAID4
- Write:** Same write performance as RAID4

### RAID4 but with spread out parity



$S_x, C_y, P_z = \text{Stripe } x, \text{ Chunk } y, \text{ Parity } z$

---

### RAID5

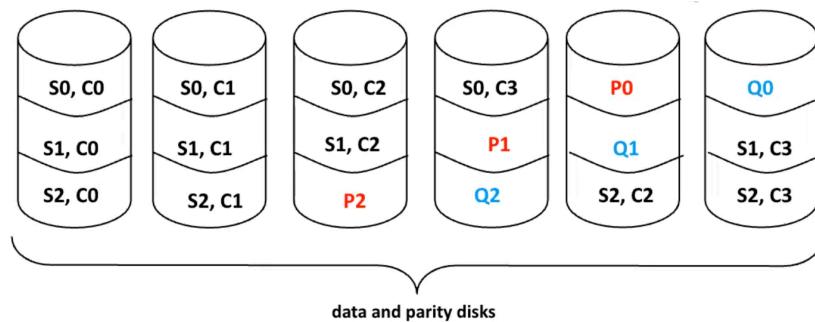
Metric	Evaluation

<b>Space Efficiency</b>	$1 - 1/N$ , taking the parity disk into account, otherwise like RAID0
<b>Storage Capacity</b>	$(N - 1) \times C$ , for $N$ disks with the same capacity of $C$ , as we're still losing the capacity equal to a whole disk on parity
<b>Fault Tolerance</b>	$F = 2$ , same as RAID4
<b>Read Performance</b>	$N$ , no longer impaired by the parity disk
<b>Write Performance</b>	$N - 1$ , we still have to write parity each time, same as RAID4
<b>Min Disks Required</b>	3, same as RAID4

## RAID6

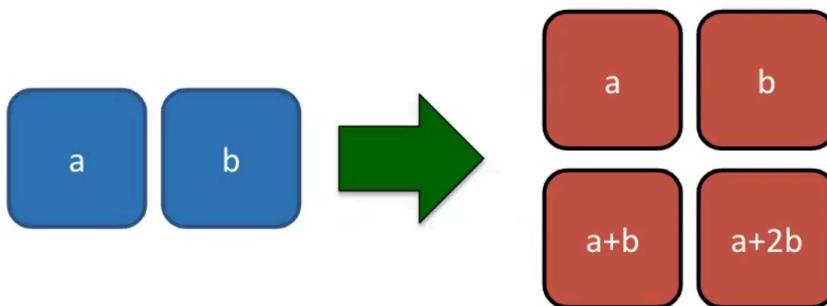
- ◆ **Read:** can outperform RAID5, as there are multiple parities to take from
- ◆ **Write:** slower than RAID4, because we need to perform more parity
- ◆ Better *reliability* than RAID4

Like RAID5 with an extra parity

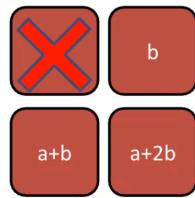


$S_x, C_y, P_z, Q_w = \text{Stripe } x, \text{ Chunk } y, \text{ Parity } z, \text{ Parity 2 (Q) } w$

- ◆ Linear combination of 2 chunks, in such a way that you don't use more space than any other chunk.
- ◆ What is  $2 \times$  bitstring?
  - ◆ Reed-Solomon Code, *see next section*
  - ◆ Possible to extent to protect against *more than 2 losses*, however, with some implications



(Reed-Solomon code)



Overhead: 100% (worst case)

Traffic to repair loss: 2 units

Protection: 2 losses



## RAID6

Metric	Evaluation
Space Efficiency	$1 - 2/N$ , same as RAID5 but taking the space of another parity disk into account
Storage Capacity	$(N - 2) \times C$ , for $N$ disks of same capacity $C$
Fault Tolerance	$F = 3$ because of the extra parity
Read Performance	$N$ , same as RAID5, but can sometimes outperform, if parity can be used from two different disks
Write Performance	$N - 2$ , taking the space of two parity disks into account
Min Disks Required	4, 2 for striping and chunking, 2 for parity

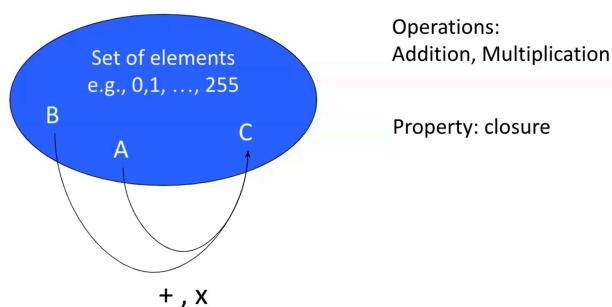
## Finite Fields

- Acceleratable with parallel GPU operations

Introduction to next lecture.

*How to set up that linear combination for extra parity*

## Closure Property



## Example GF(2<sup>2</sup>): Addition

*Instead of operating on 1 bit, operate on pairs of bits, i.e. numbers 0, 1, 2, 3*

XOR bitpair-by-bitpair

Addition				
+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

### Example GF(2<sup>2</sup>): Multiplication

*Retains some properties like multiplying with 0 always gives 0, with 1 always gives the same (identity)*

Multiplication				
+	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

### So how to do the a + 2b for the extra parity

*Using the two tables above*

$$a = 10$$

$$b = 11$$

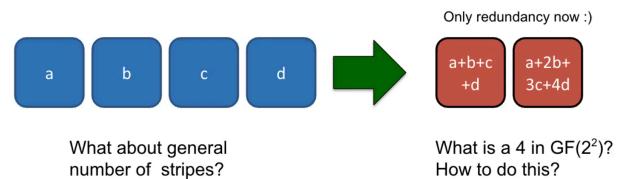
$$\begin{aligned} 2 * b &= 10 * 11 \quad (= 2 * 3) \\ &= 01 \quad (= 1) \end{aligned}$$

$$\begin{aligned} a + 2 * b &= 10 + 01 \\ &= 11 \end{aligned}$$

- ◆ *Retains the same bit-length* which is important
- ◆ Can be seen as some polynomial modulus arithmetic

### What happens with more than 3 disks

- ◆ Go to bigger bit pairs if necessary



## RAID Updating Data

### What happens if you need to modify a file?

- ◆ Deleting/inserting → Messes up striping

*If one were to touch the originally written data, then this might incur massive down-times or system slow-downs*

**Solution:**

- ◆ Changes recorded appending data never modifying the original written data

## Nested/Hybrid RAID

---

- ◆ Combine several RAID levels stacked
- ◆ Combines the different properties
- ◆ Typically uses two levels like RAID 10
  - ◆ **First level is the lowest:** RAID1 replication at the bottom
  - ◆ **Second is the highest:** Data striping of RAID0 (carried out first) on top of RAID1 replication

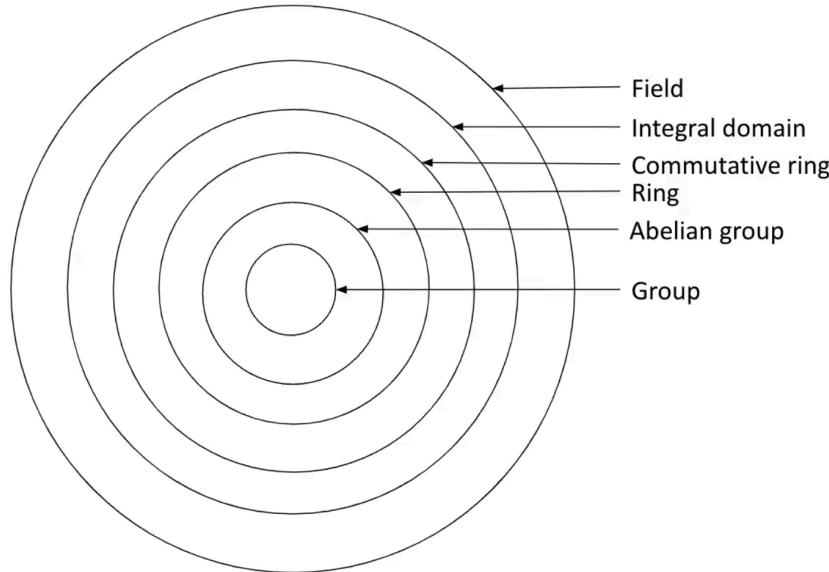
*Increases read/write speeds of RAID1 by doing the striping of RAID0*

# 05: Reliable Storage: Finite Fields & Linear Coding

## Leading up to Finite Fields

---

- ◆ Used fx. in RAID6, see previous sections
- ◆ Any operation on one or more elements from within a set of finite elements always yields an element in that set
- ◆ Any operation should always be reversible



## Groups

### Groups

A group  $G$ , denoted  $\{G, *\}$  is a set of elements with a binary operation  $*$  that associates each ordered pair  $(a,b)$  of elements in  $G$  to an element  $(a*b)$  in  $G$  following

### Axioms

- *Closure*: If  $a$  and  $b$  in  $G$ , then  $a * b$  is also in  $G$
- *Associative*:  $a * (b * c) = (a * b) * c$  for all  $a,b,c$  in  $G$
- *Identity*: Exists  $e$  in  $G$ , s.t.  $a * e = e * a = a$  for all  $a$  in  $G$
- *Inverse*: for each  $a$  in  $G$ , exists  $a'$  in  $G$ , s.t.  
 $a * a' = a' * a = e$

**Finite group:** finite number of elements

## Abelian Group

- ◆ **Group** with *commutativity*

## Rings

# Rings

A ring  $R$ , denoted by  $\{R, +, \cdot\}$  is a set of elements with two binary operations: addition, multiplication. For all  $a, b, c$  in  $R$  the following axioms are satisfied

- $R$  is **abelian group** with respect to the addition
- *Closure under multiplication:*  $ab$  in  $R$
- *Associativity of multiplication:*  $a(bc) = (ab)c$
- *Distributive laws:*  $a(b+c) = ab + ac$   
$$(a+b)c = ac + bc$$

## Commutative Ring

- ◆ **Ring** that satisfies *commutativity of multiplication*  $ab = ba \in \mathbb{R}$

## Integral Domain

- ◆ **Commutative ring** that also satisfies
  - ◆ *Multiplicative identity* e.g.  $1a = a1 = a$
  - ◆ *No zero divisors* i.e. if  $ab = 0 \Rightarrow a = 0 \vee b = 0$

## Field

- ◆ **Integral domain** that also satisfies
  - ◆ *Multiplicative inverse* i.e. for each  $\forall a \in F : a \neq 0 \Rightarrow \exists a^{-1} : a(a^{-1}) = (a^{-1})a = 1$

## Finite Fields

---

Can write fields of the form  $GF(p^n)$ , where  $p$  is prime

Addition and multiplication over  $GF(p)$  are mod  $p$

Focus on  $p = 2$

### Example:

$GF(2)$    addition: equivalent to XOR  
multiplication: equivalent to AND

**How to divide?** Multiply by multiplicative inverse

### Finding the multiplicative inverse

- 1.- Can look for  $a^{-1}$  such that  $(a^{-1} \cdot a) \equiv 1$
- 2.- Can use the extended Euclidean algorithm

- ◆  $GF(2^n)$  is very useful for binary in computer science
  - ◆ Thus works well with binary and can be *executed fast on computer*

- ◆ Much of the time *modular arithmetic* can be used to create *finite fields*
  - ◆ Because of the nature of this,  $p$  has to be a prime, if not it is no longer a finite field
    - ◆ 

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Where you can't undo, you don't know whether 1 came from  $1 \times 1$  or  $3 \times 3$ , similar case with 3.

$G(2^n)$  is *not prime*, thus we use *polynomial arithmetic* instead

### Ordinary polynomial arithmetic:

A polynomial of degree n

$$F(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 x^0 = \sum a_i x^i$$

$a_i$  are the coefficients, chosen from a set

*Operations:*

$$\text{Addition } f(x) + g(x) = \sum (a_i + b_i) x^i$$

$$\text{Multiplication } f(x) \times g(x) = \sum C_i x^i$$

$$\text{with } c_k = a_0 b_k + a_1 b_{k-1} + \dots + a_k b_0$$

- ◆ Product breaks the finite fields guarantees

- ◆ • Arithmetic of coefficients is performed modulo 2
  - i.e., using GF(2) addition/multiplication for coefficients of the same order
  - e.g.,  $(a_i + b_i) \bmod 2$
- If multiplication results in a polynomial greater than  $n-1$ , then the polynomial is reduced modulo an irreducible polynomial  $p(x)$ 
  - Think of it as a  $\bmod p(x)$  operation: divide by  $p(x)$ , keep the remainder
- ◆ Divide by some irreducible polynomial (of order  $n$ ?)

- ◆ How about  $2 + 3$ ?

$$2 = (10)_b \text{ and } 3 = (11)_b$$

As polynomials:

$$2 \equiv x \text{ and } 3 \equiv x + 1$$

Thus,  $2 + 3$  becomes

$$x + (x + 1) = 1$$

Usually you have LUT for these finite fields in an implementation, which accelerates it immensely, instead of computing these polynomials in real time

- ◆ Sometimes tables become very big like  $GF(2^8)$ , where you might *need* to compute

## How to implement multiplication GF(2<sup>n</sup>)?

### A.- Product (shifts+ XORs)

- 1) Pick one as multiplier (M) and another as multiplicand (m)
- 2) For each "1" in "M", left shift the "m" by the position of the "1"
- 3) XOR shifted versions

### B.- Modulo irreducible polynomial (long division)

- 4) Initialize: F(0) = Result of part A
- 5) Take irreducible polynomial p(x) left shift until first "1" of polynomial and of value F match
- 6) F(i+1) = (shifted p(x)) XOR (F(i))
- 7) Stop if first "1" of F(i+1) occurs in the (n-1)-th bit

$$M = (00010001)_b \text{ and } m = (10100111)_b$$

$$\begin{array}{r} 101001110000 \\ (XOR) \quad 10100111 \\ \hline 101011010111 \end{array} \quad (\text{m shifted 4 times})$$

$$\begin{array}{r} 101011010111 \\ (XOR) \quad 10100111 \\ \hline 001000001111 \end{array} \quad (\text{m shifted 0 times})$$

### B.- Modulo irreducible polynomial

$$p(x) = x^8 + x^4 + x^3 + x + 1 \quad (100011011)_b$$

$$\begin{array}{r} 101011010111 \mod p(x) \\ (XOR) \quad \underline{\textbf{100011011000}} \\ 001000001111 \\ (XOR) \quad \underline{\textbf{001000110110}} \\ 000000\textbf{111001} \rightarrow (00111001)_b \end{array}$$

## Linear Coding

---

- ◆ Generating a linear coded fragment/stripe  $C$

$$C_i = \sum a_{ij} P_j \quad (2)$$

Coded stripes/fragments

$$\left[ \begin{array}{c} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{array} \right] = \left[ \begin{array}{cccccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{array} \right] \left[ \begin{array}{c} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{array} \right]$$

$M \times M'$

*Take the original fragments  $P_j$ , and make a **linear combination** of them, resulting in the encoded fragments  $C_i$*

How you pick the coefficients determines properties and performance of the system

**RAID6**

$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \\ C_8 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{bmatrix}$$

### How to Decode (RAID6)

- You have some subset of the coded fragments, and you want to recover

#### Example:

- Lost 5 and 6

$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_7 \\ C_8 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{bmatrix}$$

Something something linear system of equations

→ *gaussian elimination*

### Codes

- MDS: Maximum Distance Separable
  - A **maximum distance separable** code, denoted by **MDS(n, k)**, has the property that any **k (< n)** out of **n** nodes can be used to reconstruct original native blocks
    - i.e., at most  $n-k$  disk failures can be tolerated
- Example:
  - RAID-5 is an MDS( $n, n-1$ ) code as it can tolerate 1 disk failure
  - RAID-6 is an MDS( $n, n-2$ ) code as it can tolerate 2 disk failures
- Reed Solomon Vandermonde: Not systematic (and MDS code)

## Reed Solomon - Vandermonde

- Not systematic  $n = 6, k = 4$
- Need GF(2^3) (at least) - primitive polynomial:  $x^3+x^2+1$
- What about the exponents?

$H = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 2^2 & 3^2 & 4^2 \\ 1 & 2^3 & 3^3 & 4^3 \\ 1 & 2^4 & 3^4 & 4^4 \\ 1 & 2^5 & 3^5 & 4^5 \end{pmatrix}$	Mapping to polynomial 0 → 0 1 → 1 2 → x 3 → x + 1 4 → x^2 5 → x^2 + 1 6 → x^2 + x 7 → x^2 + x + 1	Mapping to polynomial $2^2 \rightarrow x^2$ $2^3 \rightarrow x^3 \rightarrow R(x^3/x^3+x^2+1) \rightarrow x^2+1$ $2^4 \rightarrow x^4 \rightarrow x(x^3+1) \rightarrow x^2+x+1$ $2^5 \rightarrow x.x^4 \rightarrow x(x^2+x+1) \rightarrow x + 1$ $2^6 \rightarrow x.x^5 \rightarrow x(x+1) \rightarrow x^2+x$ $2^7 \rightarrow x.x^6 \rightarrow x(x^2+x) \rightarrow x^3 + x^2$	→ 4 → 5 → 7 → 3 → 6 → 1
--	---	--	--

- ◆ Reed Solomon Cauchy: Systematic and very constructive (designed for specific purposes)

- ◆ Many Cauchy alternatives

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 3^{-1} & 4^{-1} & 5^{-1} \\ 3^{-1} & 1 & 6^{-1} & 7^{-1} \end{pmatrix}$$

## Reliable Multi-server

---

- ◆ Numerous disk *failures per day*
- ◆ Failures are the *norm*
- ◆ Redundancy is *necessary*
- ◆ Replication or erasure coding

### ⊗ Question

Are RAID schemes like RAID5 and RAID6 a form of *erasure coding*?

### ⊗ Possible Report Weakness

**Erasure coding** and **replication** are two *distinct* methods of redundancy, there are trade-offs for each.

With **erasure coding** you typically have to do *more work* to recover.

→ a lot of network traffic for repair

Where with simple **striped replication** you might only have to move 1/10 a file.

→ little network traffic for repair

Apparently *regenerating codes* are codes that keep the MDS properties and reduce repair traffic.

## ☒ Possible Report Weakness

I was *definitely* supposed to use PyErasure for the final project, which I didn't  
→ look at Lab Week 5

# 06: Repair Problem & Regenerating Codes

## Repair Problem

- ◆ See intro from last week, in terms of why and trade-offs
- ◆ Last week talked about erasure codes; striping with redundancy, e.g. RAID-like, array codes
- ◆ This week will be about *regenerating codes*, optimised for repair; decoding and minimising network traffic

### ☒ Possible Report Weakness

Did I say that the loss-quantification ping/pong was implemented in the Lead Node, without actually having it implemented there?

## Regenerating Codes

- ◆ Stripe data with redundancy and reduce the repair bandwidth
- ◆ Minimise repair traffic with same fault tolerance as Reed Solomon

### Benefits:

- ◆ Faster repair
- ◆ Inherent trade-off of storage and traffic-cost
- ◆ **Network coding is key:** Recoding needed at the nodes/racks before sending

*Don't need to become expert in network coding*

→ **Just know:** Network itself can provide re-combinations of the coded data

## Repair Degree

Repair degree  $d$

- ◆ That is, from how many disks  $d$  data is needed to repair.

## More General Reed Solomon

How to implement multiplication  $\text{GF}(2^n)$ ?

### A.- Product (shifts+ XORs)

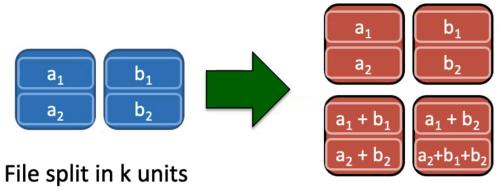
- 1) Pick one as multiplier ( $M$ ) and another as multiplicand ( $m$ )
- 2) For each “1” in “ $M$ ”, left shift the “ $m$ ” by the position of the “1”
- 3) XOR shifted versions

### B.- Modulo irreducible polynomial (long division)

- 4) Initialize:  $F(0) = \text{Result of part A}$
- 5) Take irreducible polynomial  $p(x)$  left shift until first “1” of polynomial and of value  $F$  match
- 6)  $F(i+1) = (\text{shifted } p(x)) \text{ XOR } (F(i))$
- 7) Stop if first “1” of  $F(i+1)$  occurs in the  $(n-1)$ -th bit

## Example

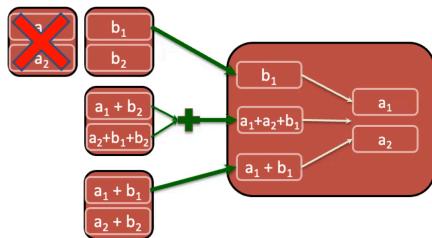
Generate:



Overhead: 100%

#### Recover:

- Only had to move 1.5 units instead of 2



Overhead: 100%

Protection: 2 losses of  
nodes/racks

#### How

- as  $n$  and  $k$  refer to internal units here and not specific nodes you can determine how thinly to slice/stripe for optimal behaviour

For a file of size  $B$  bits, split into  $k$  pieces and encoded  
into  $n$  pieces

Each piece is stored in a disk with  $B/k$  bits per disk

**Theorem [Vanilla]:** it is possible to **functionally repair a code by communicating**

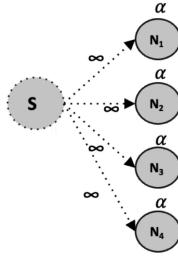
$$\frac{n-1}{n-k} \frac{B}{k} \text{ bits}$$

As opposed to transmitting  $B$  bits (naive cost)

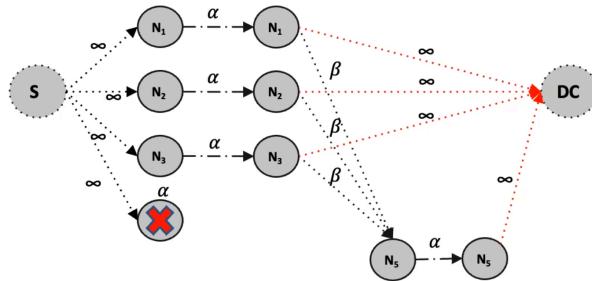
**Previous Example:** For  $n = 4$ ,  $k = 2$ , then at least  $\frac{3}{4} B$  bits are needed (and we used exactly that)

#### Graph

- Multicast flow problem
- Each node stores  $\alpha$  bits
- Original file has  $B$  bits
- Spread is  $\alpha = B/k$  bits



- Transmit  $\beta$  bits from each surviving node ( $d$  surviving nodes)
- Total bandwidth of  $\gamma = d\beta$
- As long as there is a linear combination that brings the *correct* properties, it is solvable
- Maintain MDS conditions



## ☒ Possible Report Weakness

Kinda have to do well to make up for report weaknesses it seems like

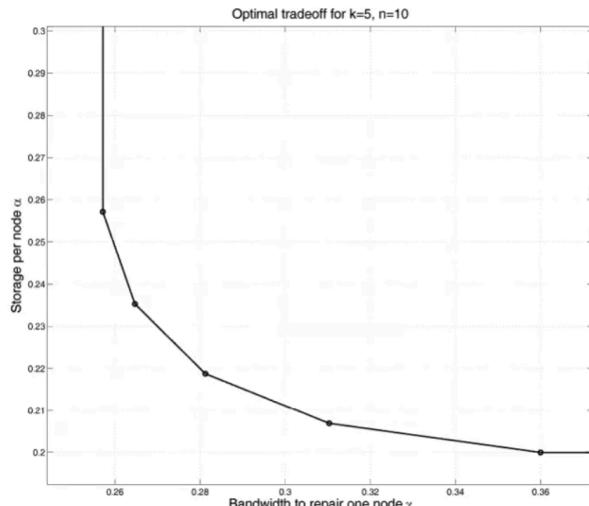
### Full theorem:

- Clear connection between the degree  $d$  and other parameters

$$\alpha(n, k, d, \gamma) = \begin{cases} B/k & \gamma \in [f(0), +\infty) \\ \frac{B - g(i)\gamma}{k - i} & \gamma \in [f(i), f(i - 1)) \end{cases}$$

$$f(i) = \frac{2Bd}{(2k - i - 1)i + 2k(d - k + 1)}$$

$$g(i) = \frac{(2d - 2k + i + 1)i}{2d}$$



- Hard limit at some point
  - $\rightarrow \alpha \approx 0.258$  for minimum bandwidth
  - $\rightarrow \alpha \approx 0.2$  for minimum storage

### Minimum Bandwidth Regenerating (MBR):

$$(\alpha, \gamma) = \left( \frac{2Bd}{2kd - k^2 + k}, \frac{2Bd}{2kd - k^2 + k} \right)$$

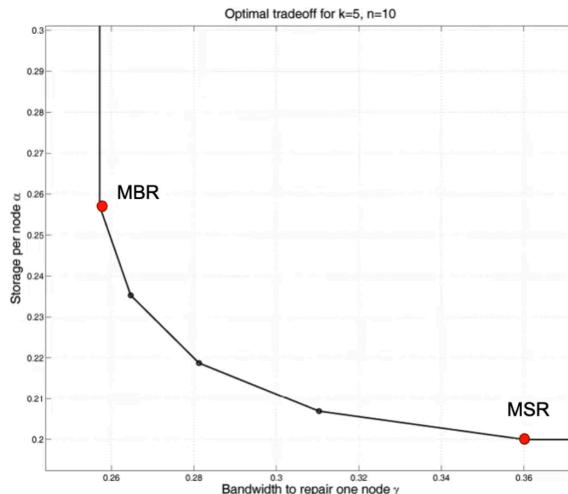
Consider case of  $d = n - 1$

### Minimum Storage Regenerating (MSR):

$$(\alpha, \gamma) = \left( \frac{B}{k}, \frac{Bd}{k(d - k + 1)} \right)$$

Case of  $d = n - 1$  was the 'vanilla' theorem of before

**Caveat:** What is hidden behind this result?



- Assuming granularity at bit-level, that you can slice anywhere at bit-level, which might not be possible.  
→ Can't get exactly MBS or MSR in these cases

### Beyond Full Node Loss

- What happens if we only lose one or several disks on a node, instead of all of it

#### ☒ Possible Report Weakness

Not a weakness but the report doesn't talk about node vs inter-node disk loss

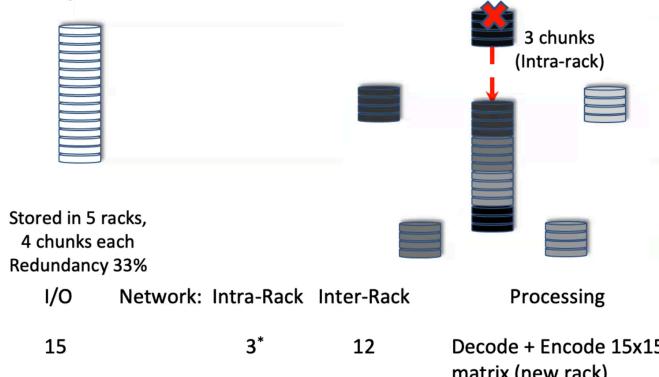
### RS vs RLNC

#### Example: Reed Solomon

Collect 15 chunks → decode → re-encode

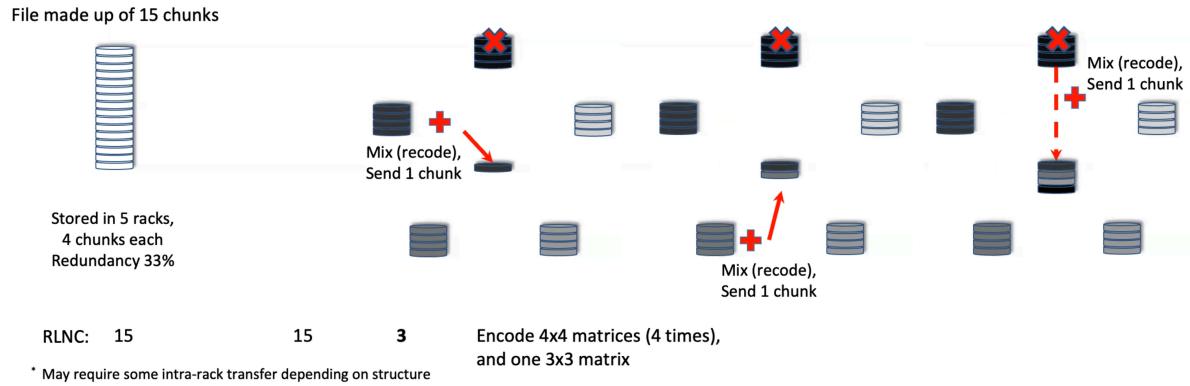
Centralised processing at one node

File made up of 15 chunks



RLNC:

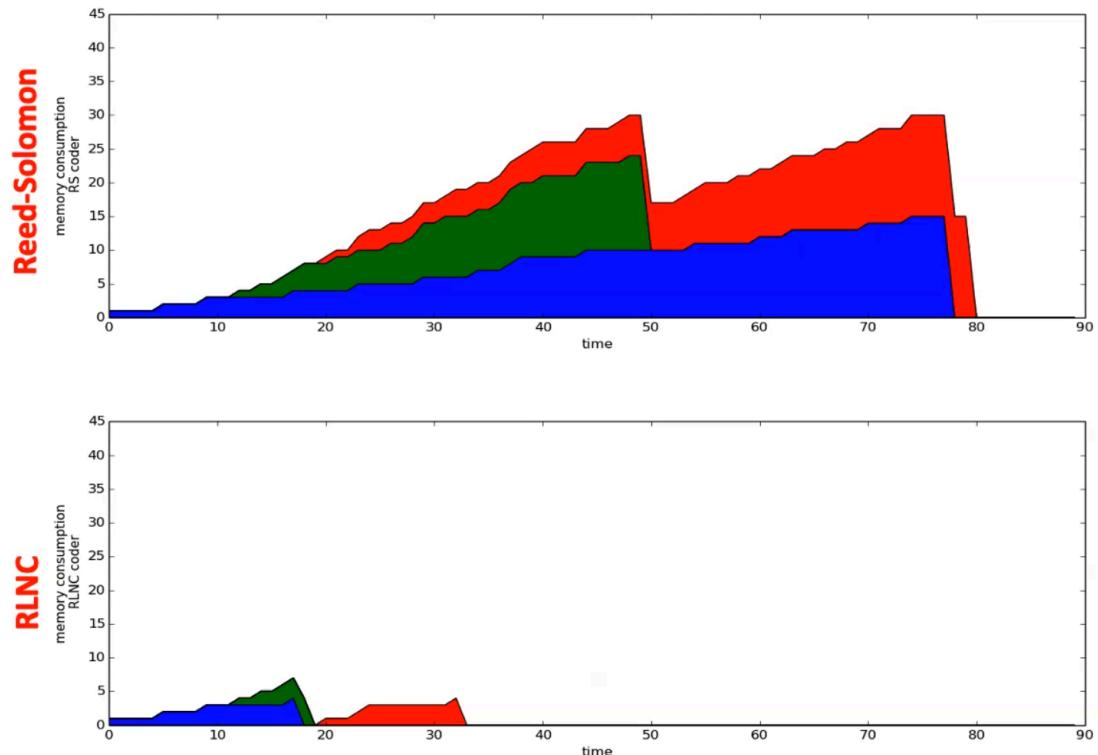
#### Example: Random Linear Network Code Distributing the processing between nodes



### Benefits:

- ◆ Less memory consumption during recovery process
- ◆ Workload can be *distributed* during recovery process (across racks/nodes), while **Reed Solomon** will have large workload on a single rack/node at the end
- ◆ Less delay after last packet arrives for **Random Linear Network Codes**

## Memory Consumption RS vs RLNC



### ☒ Possible Report Weakness

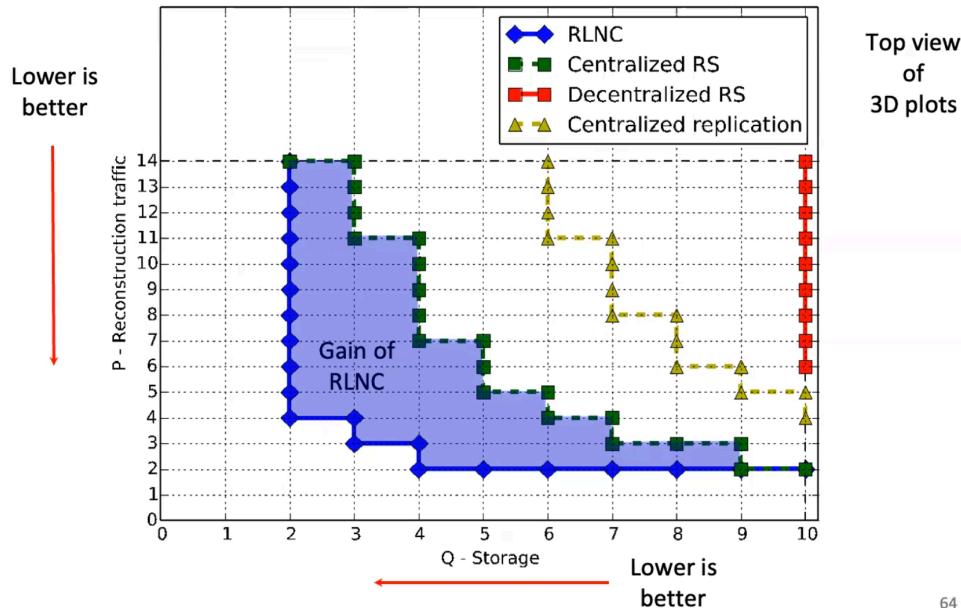
I don't fully understand how RLNC is able to cut down traffic so much?  
→ does it mean something about doing it in the network instead of on the nodes?

#### Answer:

- ◆ Combine multiple packets of data into a single coded packet of the same size.
- ◆ Coded packets are not useful without the coding coefficients

### Over best alternative, NC buys you:

- Up to 65% reduction in storage
- Up to 71% reduction in network use



64

### Exact Repair

**E-MBR (Minimum bandwidth regenerating codes)** with exact repair:

- ◆ Minimise repair bandwidth while maintaining MDS property

**Idea:**

- ◆ Assume  $d = n - 1$  (repair data from  $n - 1$  survival disks during a single-node failure)
- ◆ Make a duplicate copy for each native/code block
- ◆ To repair a failed disk, download *exactly one block per segment* from each survival disk

**Feasibility:**

- ◆ Achieving exact repair comes at a cost
- ◆ Not really feasible
- ◆ Requires more control with a specific structure

*Needs something close to the Genie policy*

### ☒ Possible Report Weakness

Gives no perspective to repairability at all!

In general a lack of giving *perspectives* to the rest of the course topics

# 07: Reliable Storage: Regenerating Codes & Local Repairability

## Exact Repair & Regenerating Codes

- ◆ Introduction to this from last week

### Functional repair vs exact repair

#### What

**Exact:** You get exactly what you lost

- ◆ Reed Solomon
- ◆ RAID systems

**Functional:** Can be used for the same purpose, but aren't exactly the same values (like multiplications of data?)

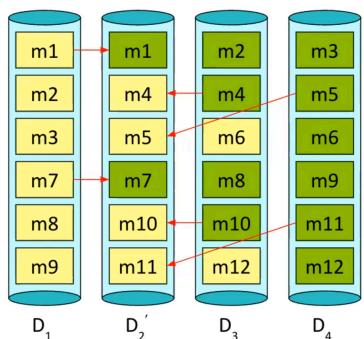
- ◆ Regenerating codes (RLNC)

### How to do Exact Repair with RLNC

*MBR and MSR points are predicated on functional repair*

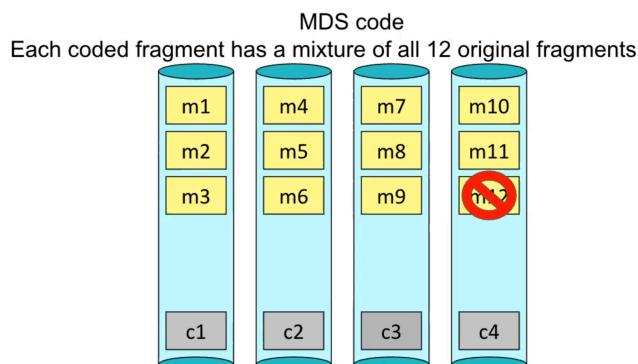
RLNC can still be exact repair, but that restricts it in many ways, making it less efficient.

- ◆ Fx. E-MBR
  - ◆ Duplicate each native block. Native and duplicate blocks are in different disks



If D<sub>2</sub> had been lost, it can be repaired easily ↑

- ◆ Fx. Exact repair with known storage structure (kinda MSR)



“Interference Cancellation”

If we know  $c_4 = m_1 + 2 \times m_2 + 3 \times m_3 + \dots + 11 \times m_{11} + 12 \times m_{12}$

## Local Repairability

- ◆ XORBAS
- ◆ Locality in terms of within same coding?
- ◆ Placement is important

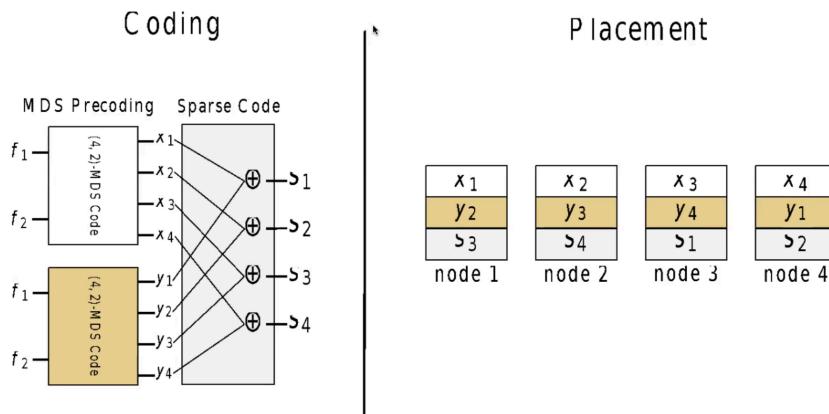
**Theorem 1** For locality  $r$ , there is a related storage cost

$$(r, \gamma) = \left( r, \alpha_{MDS} \left( 1 + \frac{1}{r} \right) \right)$$

**Theorem 2** This is the optimal trade-off between locality and storage

### Example

We want to have locality

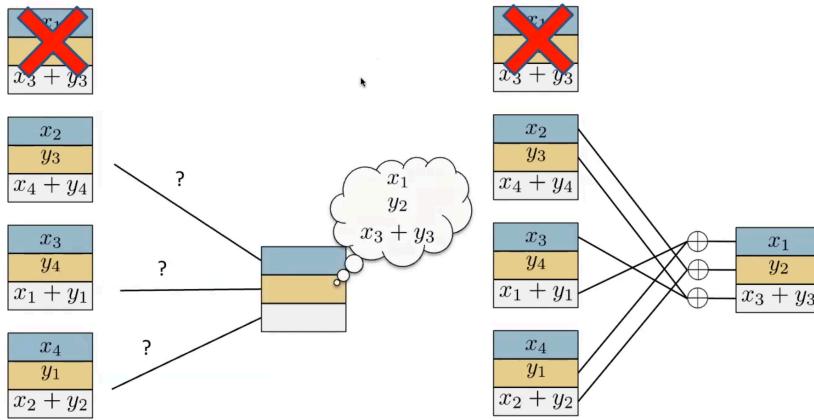


- ◆ If you keep all involved in one code,  $x_1, y_1, S_1$  on the same node, then you will lose that data entirely.
  - ◆ Sparse combinations and its parents *have* to be stored on different nodes
- ◆ With simple clever placement, you can avoid this issue
- ◆ Can sustain 2 full node losses, without losing data
  - ◆ Due to sparse extra combinations

### ☒ Possible Report Weakness

Require better intuition about MDS codes

### How to repair

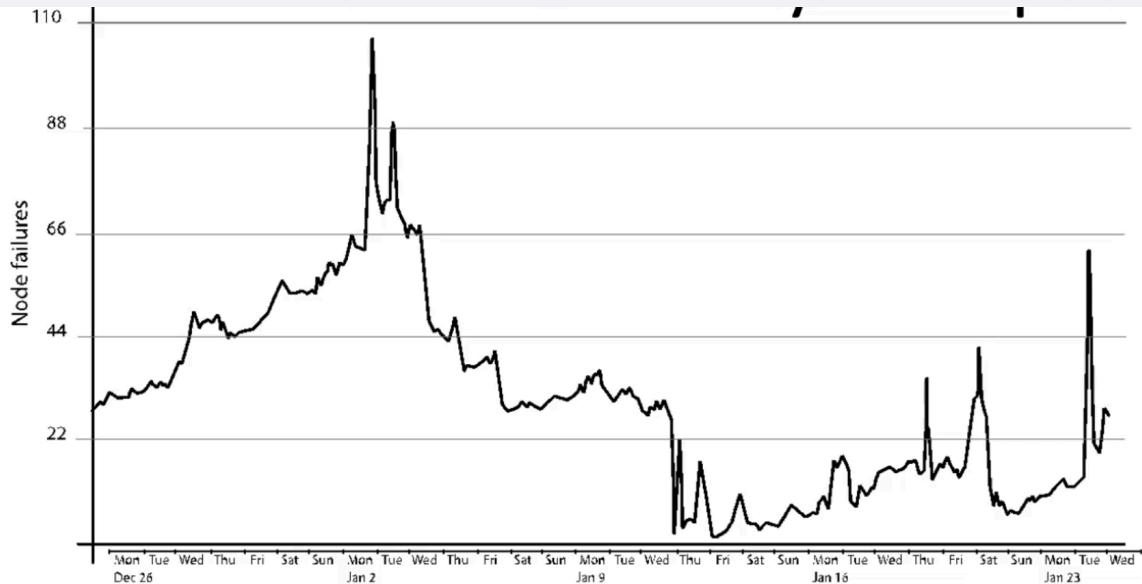


- ◆ Achieves the optimal performance from theorem 1 & 2 above ↑

## ☒ Possible Report Weakness

He likes the problem from the Facebook setup.

→ The *big picture* view



20 node failures \* 15TB = 300TB

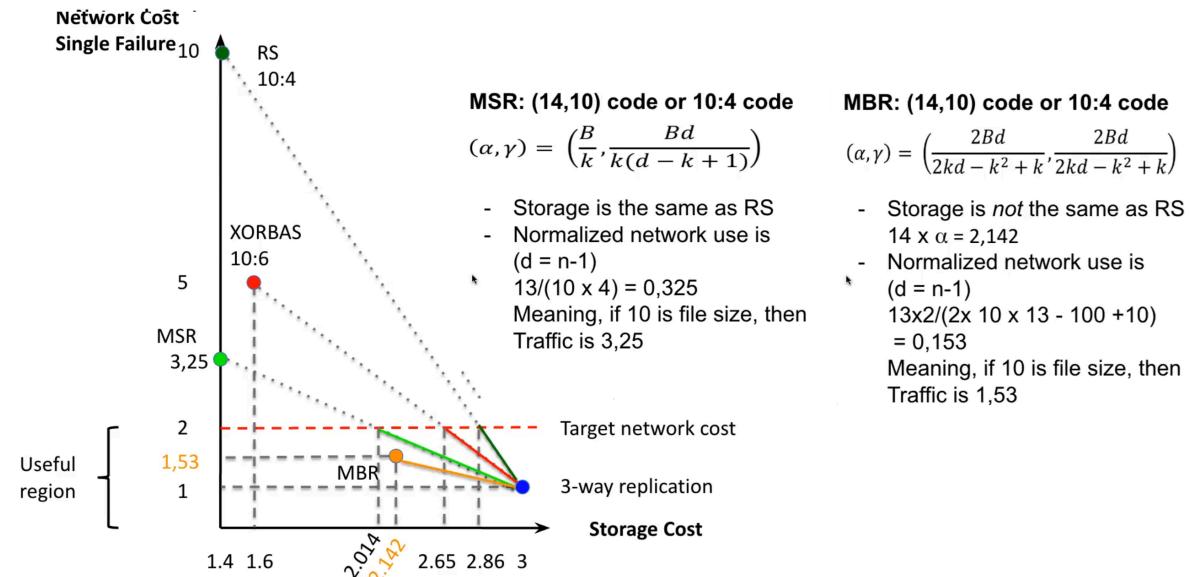
if 8% RS coded, 588TB network traffic/day. (average total network: 2PB/day)

**~30% of network traffic is repair in a normal day**

*Motivation* ↑ → Make sure that network traffic doesn't exceed a certain level

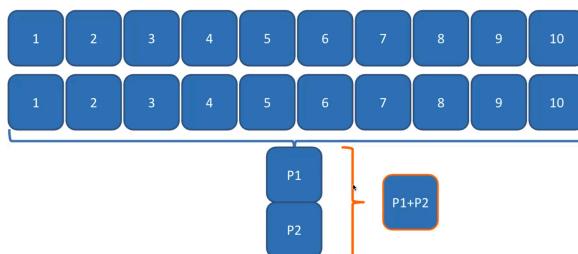
## How Good

- ◆ **Reed Solomon 10:4:** 3x storage → 2.865x storage = 5% improvement
- ◆ **XORBAS 10:6:** 3x storage → 2.65x storage = 13% improvement (25% erasure coded)
  - ◆ Still paying a hefty cost in traffic



## Hybrid Replication & Coding

- ◆ Not good from an erasure coding perspective
- ◆ Quite restricted
  - ◆ Lost 8, 9, 10 → doomed
- ◆ Not nearly as flexible as RS and MDS
- ◆ Compromising structure
- ◆ *Not* erasure coded



**Storage cost:** 2.3 stored files per original file

**Network use (recover 1 loss):**  $26/23 \sim 1.13$  file

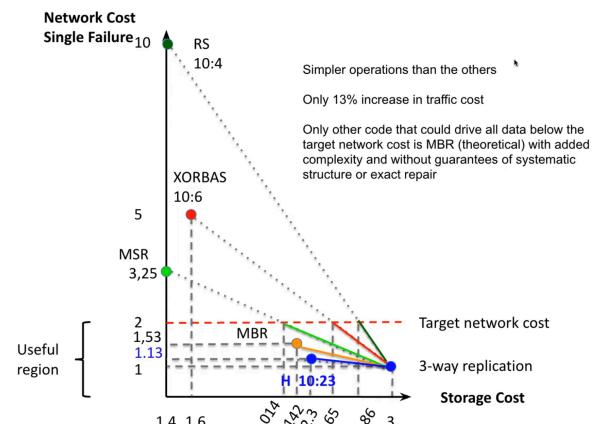
**Reliability:** Recovery from at least 4 losses

**Processing:** encoding 1 file when losing P1, P2, or P1+P2

## Can this help?

Maintaining network use to roughly 2 files per lost file as an acceptable measure

- We could in principle have all files in this coding
  - Minimal increase in bandwidth for repair (13% wrt 3-way replication)
  - Higher reliability than 3-way replication
  - Same or higher reliability as RS 10:4 code
  - May hurt availability slightly
  - Very low processing requirement most of the time
- Storage reduction? From 3x to 2.3x → 30.4%!
  - Merely determined by availability
  - Now we are making a dent (from 5% to 30.4%)



## Is it “silly”?

1. Network use under control
2. Storage reduction:
  - 10:4code → up to 5%
  - XORBAS → up to 13%

Best “Silly” approach
 
  - Savings of 30.4% with higher reliability than 10:4 code
  - Only 13% traffic increase for the loss of 1 unit
3. More important: the key is to make a mix of the various codes, so far mixed 2...horizon is open to richer mixtures

# 08: Hadoop Distributed File System (HDFS)

## General

---

- ◆ Use ideas from previous weeks

## Purpose

- ◆ Process *big data* with reasonable *cost and time*
- ◆ Large scale processing on data not stored on a single computer

## Idea

- ◆ Distribute the data as it is initially stored.
- ◆ *Each node* can then perform computation on the data it stores without moving the data for the initial processing.
- ◆ **Don't** move data to the computation
- ◆ **Do** move the computation to where the data is

## Examples

- ◆ Google File System
- ◆ MapReduce: Simplified Data Processing on Large Clusters

## History

Used to be → is like this now

- ◆ Low data volume → High data volume
- ◆ Big complex computations → Many simpler computations

*Improvements to the individual machine* → *able to perform the processing itself*.

- ◆ Moore's law

## Hadoop File system

---

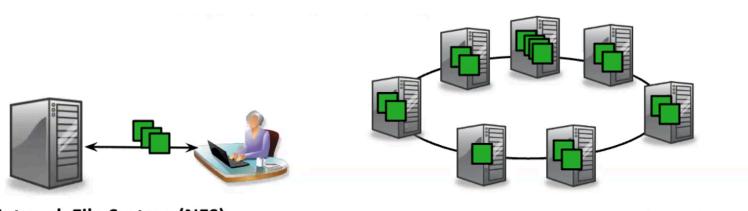
- ◆ Storing data on the cluster
- ◆ For *large* files
- ◆ Files *split into blocks* of certain size → distribute across nodes in cluster
  - ◆ Similar concept as we saw in RAID
- ◆ Each *block is replicated* multiple times
- ◆ Main goal to *optimise for speed*
- ◆ Has some *reliability*, but not a main goal
- ◆ Unlikely to be built with SSDs → too expensive

### ☒ Possible Report Weakness

OH?! The storage system I have implemented is a simple HDFS??

## NFS vs HDFS

NFS < v4.1 - before multiple servers



**Network File System (NFS)**

- Single machine makes part of its file system available to other machines
- Sequential or random access
- PRO: Simplicity, generality, transparency
- CON: Storage capacity and throughput limited by single server

**Hadoop Distributed File System (HDFS)**

- Single virtual file system spread over many machines
- Optimized for sequential read and local accesses
- PRO: High throughput, high capacity
- "CON": Specialized for particular types of applications

## Basic concepts

- ◆ Redundant storage for MASSIVE amounts of data
- ◆ HDFS works best with a *small number* of *large files*
  - ◆ Millions as opposed to billions of files
  - ◆ Typically 100MB or more per file
  - ◆ Terabytes or Petabytes of data
- ◆ Files in HDFS are write once!
- ◆ Optimised for streaming reads of large files and not random reads → operate disks at absolute top speed

## More

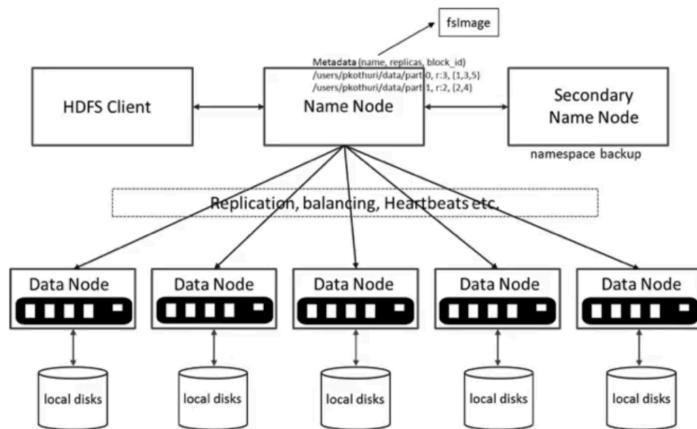
- ◆ *Very large* distributed file system
  - ◆ 10k nodes, 100 million files, 10PB
- ◆ Don't assume *specific* hardware: Assumes commodity hardware
  - ◆ Files are *replicated* to handle hardware failure
- ◆ **Optimised for batch processing**
  - ◆ Data locations exposed so that *computations can move to where data resides*
  - ◆ Provides very *high aggregate bandwidth*

## More more

- ◆ Tolerate node failure without data loss
  - ◆ *Detect failures and recover from them*
- ◆ Data coherency
  - ◆ Write-once → read-many access model
  - ◆ Client can *only* append to existing files
- ◆ *Computation is near the data*
- ◆ Portability; built using Java

## Summary

- HDFS is good for
  - Very large files
  - Streaming data access
  - Commodity hardware
- HDFS is not good for
  - Low-latency data access
  - Lots of small files
  - Multiple writers, arbitrary file modifications
- ◆ Built for large scale processing, not quick access



## Blocks

### Large blocks

- ◆ Size of either 64MB or 128MB
- ◆ Minimise seek costs
- ◆ **Benefit:** can take advantage of any disks in the cluster
- ◆ Simplified the storage subsystem: amount of metadata storage per file is reduced
- ◆ Good for replication schemes

## Master/worker pattern

### Name/Coordinator Node

- ◆ Manage filesystem metadata
- ◆ Where are chunks corresponding to a file
- ◆ Configuration
- ◆ Replication engine

### Data/Worker nodes:

- ◆ Workhorse of the file system: store and retrieve
- ◆ Stores metadata of a block (e.g. CRC checksums)
- ◆ Serves data and metadata to clients

### Data/Worker nodes:

- ◆ Two files:
  1. Data
  2. Metadata: checksums, generation stamp (timestamp?)
- ◆ Startup handshake
- ◆ After handshake

## Namenode metadata

*In memory*

- ◆ List of files
- ◆ List of blocks for each file
- ◆ List of datanodes for each block
- ◆ File attributes, e.g. creation time, replication factor
- ◆ Transaction log

## ☒ Possible Report Weakness

Well it looks like it is a very very abstract and simple implementation of HDFS if anything.

Some things line up:

- ◆ Replication schemes
- ◆ Master/worker pattern
- ◆ Coordinator keeps track of metadata

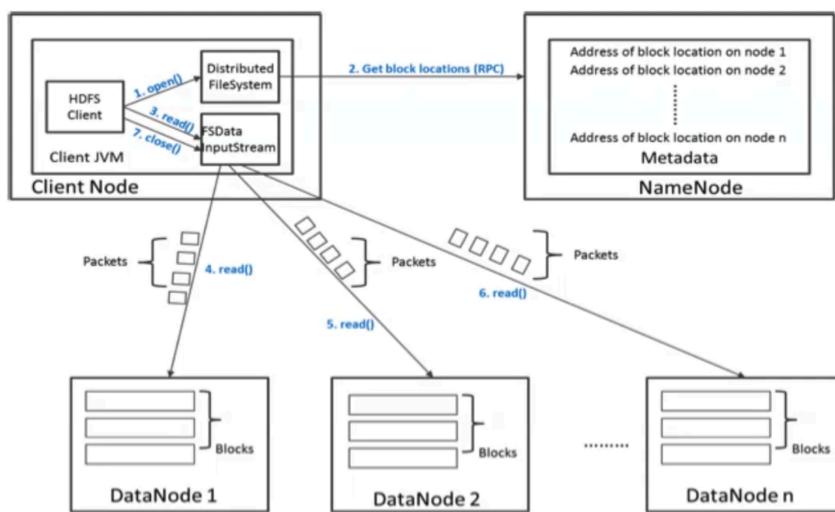
## Resiliency

Have secondary coordinator/name-node

## HDFS Client

- ◆ Code library that exports the HDFS interface
- ◆ Allows user application to access the file system
- ◆ **Intelligent client**
  - ◆ Client can find location of blocks
  - ◆ Client accesses data directly from worker/data-nodes

## Read Operation



## ☒ Possible Report Weakness

### Difference to my implementation:

*Clearly* my client does not simply use the Name/coordinator-node to access metadata and then connect to data/worker-nodes itself.

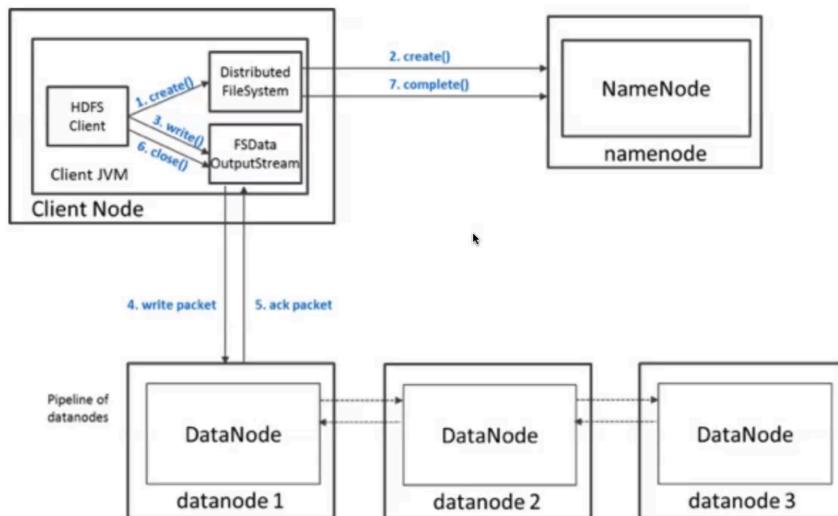
Make sure to understand whether what I have done is *RPCs* or not  
 → Maybe it's a simplified custom way of doing RPCs

#### **Remember:**

It said nowhere on the project description that it should be a HDFS implementation

### **Write Operation**

- ◆ Once written, cannot be altered, *only append*
- ◆ HDFS Client: lease for the file
  - ◆ Renewal of lease
  - ◆ Lease: *soft* limit or *hard* limit
- ◆ Single-writer, multiple-reader model



- ◆ Bring traffic down by letting the nodes talk to each other
- ◆ Circumvents network limitation on client side, as instead of talking to multiple nodes, it just sends once, thus *handing off* the network load to the nodes

### **☒ Possible Report Weakness**

My data nodes do not talk to each other at all  
 → All happens from the coordinator to the data nodes

### **Replication**

- ◆ Multiple nodes for reliability
  - ◆ Additionally, data transfer bandwidth is multiplied
- ◆ Computation is near data
- ◆ Replication factor, default:  $k = 3$

#### **Block placement:**

- ◆ After placement you have a certain structure, and knowldedge thereof
- ◆ Place intentionally to be robust and reliable
  - ◆ Spread across multiple racks
  - ◆ Nodes of rack share switch
- ◆ Balance with speed

### Balancer:

- ◆ Balasnce disk space usage
- ◆ Optimise by minimising inter-rack copying

### Strategy:

- ◆ Current Strategy
  - ◆ One replica on local node
  - ◆ Second replica on remote rack
  - ◆ Third replica on same remote rack
  - ◆ Additional replicas are randomly placed
- ◆ Clients read from nearest replicas
- ◆ Would like to make this policy *pluggable*?
  - ◆ Something you can edit and change.

## Data Pipelining

- ◆ Client retrieves list of data/worker-nodes on which to place replicas of a block
- ◆ Client writes block to the first data/worker-node
  - ◆ This node handles handing over to the next node
- ◆ When all replicas are written, the client moves on to write the next block in file

## Balancer

### Goal:

- ◆ Percentage full on data/worker-nodes should be *balanced*

### Interesting when:

- ◆ New nodes are added
- ◆ Rebalancing? Maybe when new nodes are added or simply slowly balance by using that node more.
- ◆ Rebalancer gets throttles when more network bandwidth is needed elsewhere
- ◆ Typically a command line tool run by and operator

## Block Scanner

- ◆ Periodically scan and verify checksums
- ◆ Check for:
  - ◆ Verfication succeeded?
  - ◆ Corrupt block?
- ◆ Verfication of data correctness

## Checksums

- ◆ Computs CRC32 for ever 512 bytes
- ◆ Make sure nothing is missing
- ◆ Data/worker nodes store checksums
- ◆ File Access:
  - ◆ Client retrieves the data and checksums from the data/worker-node
  - ◆ If *validation fails* → client tries other replicas

## ☒ Possible Report Weakness

No checksums, data correctness is in my system

## Heartbeat

- ◆ Data/worker-nodes sends heartbeat to name/coordinator-node
  - ◆ once every 3 seconds
- ◆ Name/coordinator-node uses heartbeats to detect whether a node has failure

## Beyond One Node Failure

### A lot of failures:

- ◆ Full racks
- ◆ Fire
- ◆ Power outages

{HDFS, GFS, Windows Azure, RAMCloud} uses random placement

→ Random not so good no no no

## Copysets & Random Replica Placement

### ☒ Possible Report Weakness

Yay finally something in the report, I must already know this.

But here are notes on what I missed or maybe *misinterpreted* from the slides:

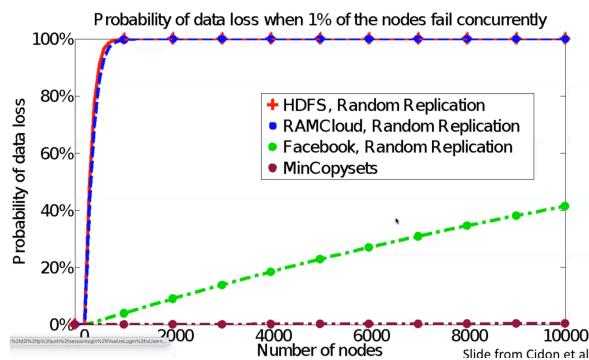
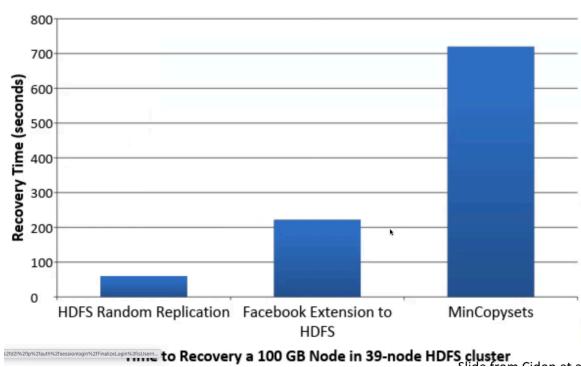
### 1% of nodes fail concurrently:

- ◆ Chance of data loss doesn't mean that percentage of data is lost
- ◆ It is a percentage of the probability of data loss, so at least 1 chunk lost

	MinCopysets	Random Replication
Mean Time to Failure	625 years	1 year
Amount of Data Lost	1 TB	5.5 GB

- 5000-node cluster
- Power outage occurs every year

Confirmed by:



## Random

- ◆ Basically every failure is a loss

Probability of a given chunk is lost:

$$\frac{\binom{F}{R}}{\binom{N}{R}} \quad (3)$$

Where  $F$  nodes out of  $N$  have been lost, with replication factor  $R$

### Minimum Copysets

- ◆ Only loss if all nodes in specific copyset dies
- ◆ Once you lose something, you will lose *a lot*
  - Less nodes left to recover (only the last nodes left in the copyset, instead of from many random nodes)
  - Thus much longer recovery time

### ☒ Possible Report Weakness

Scatter-width?

→ Daniel didn't mention in the lecture

### Buddy

- ◆ Combining random with the idea of creating separate groups
  - ◆ Isolated random replication groups
  - ◆ A copyset is always within one buddy group
  - ◆ However, placed randomly within the group

## MapReduce

- ◆ Fast overview

### Overview

- ◆ Distribute computation across nodes

### Features

- ◆ Automatic parallelisation and distribution
  - Bad for little data, good for massive data
- ◆ Provides a clean abstraction for programmers to use

### Fault Tolerance:

- ◆ System detects laggard tasks
- ◆ Speculatively executes parallel tasks on the same slice of data

### Phases

1. Map
2. (Shuffle & Sort)
3. Reduce

### Mapper:

1. **Input:** Data *key/value pairs*
  - ◆ The key is often discarded
2. **Outputs:** *zero or more* key/value pairs

- ◆ For every key identify what the value is

### Shuffle & Sort:

- Output from the mapper is *sorted by key*

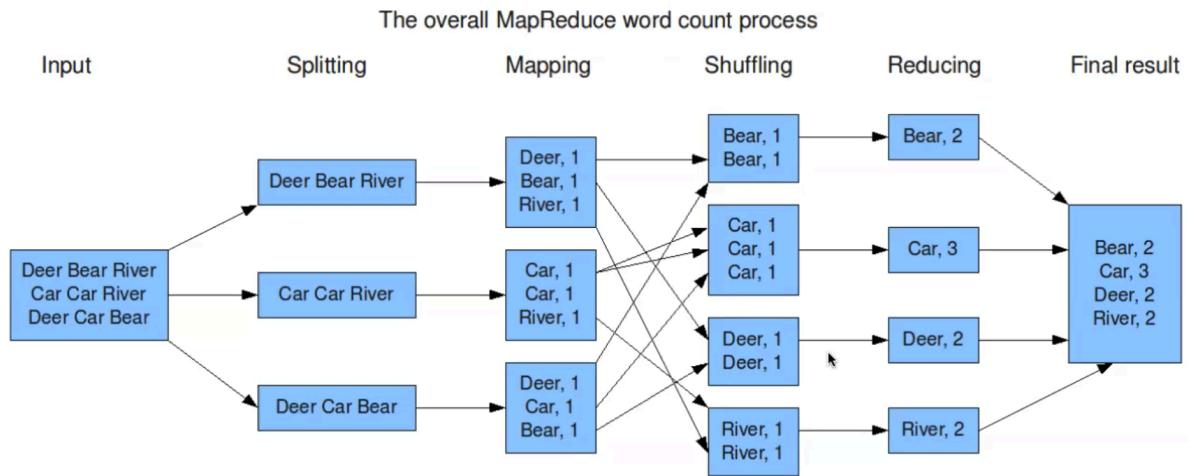
*All values with the same key are guaranteed to go to the same machine*

→ Hadoop overhead here?

### Reduce:

- Called once for each unique key
- Input:** List of all values associated with a key
- Outputs:** Zero or more final key/value pairs
  - ◆ Usually reduced to 1 output per unique key input

**Example:** Count words of same type



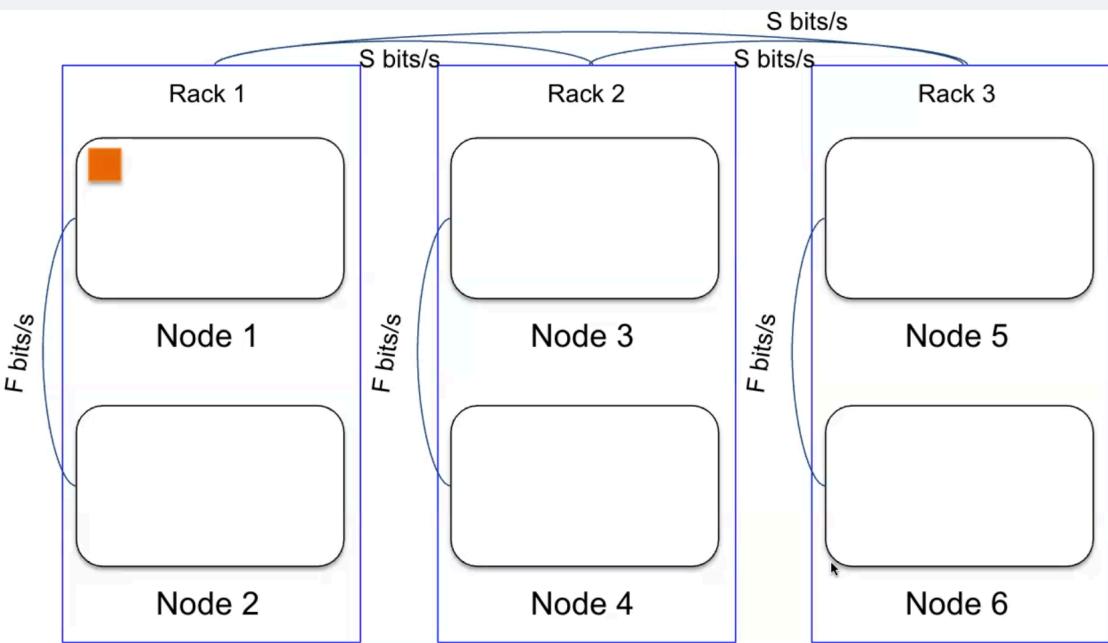
- ◆ Already at the splitting stage when stuff has been chunked and split into nodes.
- ◆ What the mapping and reduction does is defined base on what function you want to compute.
- ◆ In this case the mapping could actually also reduce the data by counting each value up, if a key is repeated.

Steps:

- Mapping:** Take each word and write it's value of 1 such that you can count them in reduction stage.
- Shuffle:** Move data around such that data for a single key is in a single node.
- Reduction:** Here reduce over each key bu summing values.

### ☒ Possible Report Weakness

This definitely looks like something I should have thought about in the report:



- 1 - How would Hadoop create 3 replicas in total? Write all set of nodes that could be selected
- 2 - Time to replicate a file of  $D$  bits From Node 1 using of the sets in (1) - Called  $T_1$
- 3 - Considering that replicas need to be generated in Node 3 and Node 5,
  - Compute the time to replicate ( $T_2$ )
  - Compute the Gain of Hadoop w.r.t. this strategy :  $G = T_2/T_1$
- 4 - If we want to generate 4 replicas in total, what would be the Average time to perform the replication considering Hadoop's strategy. Consider that no replicas are deployed in the same node

# 09: Storage Virtualisation

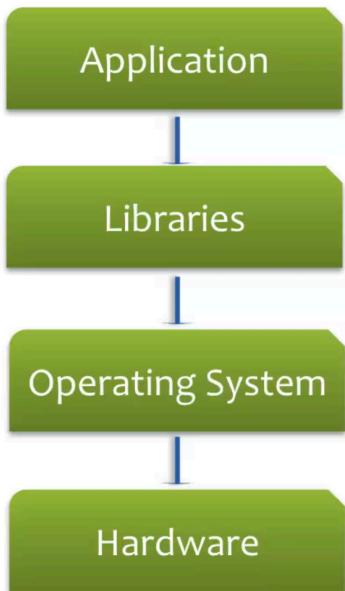
## ☒ Possible Report Weakness

Explore allocation problems!

- Which I have done
- How the system responds to node losses
- Which I have also done

## Storage Virtualisation

Machine Stack showing virtualization opportunities



- Creation of a virtual version of hardware using software
- Runs several applications at the same time on a single physical server by hosting each of them inside their own virtual machine
- By running multiple virtual machines simultaneously, a physical server can be utilized efficiently

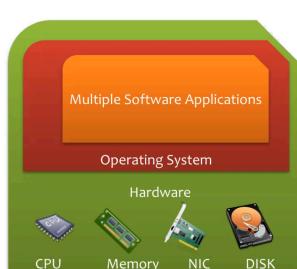
Primary approaches to virtualization

- Platform virtualization      Ex : Server
- Resources virtualization    Ex : **Storage, Network**

## Hypervisor

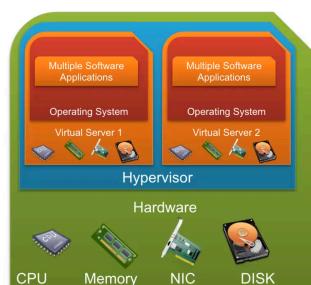
Enables the capability of manipulating and running several OS on the same hardware.

### Server without Virtualization



- Only one OS can run at a time within a server
- Under utilization of resources
- Inflexible and costly infrastructure
- Hardware changes require manual effort and access to the physical server

### Server with Virtualization

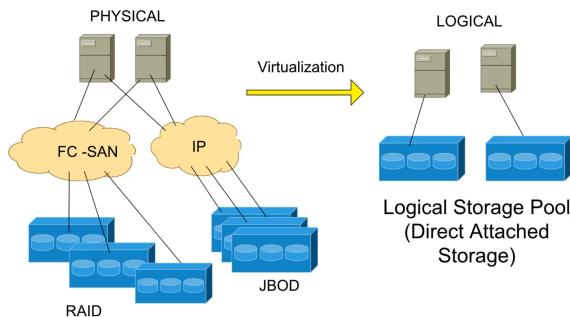


- Can run multiple OS simultaneously
- Each OS can have different hardware configuration
- Efficient utilization of hardware resources
- Each virtual machine is independent
- Save electricity, initial cost to buy servers, space
- Easy to manage and monitor virtual machines centrally

## Virtualisation Solves

- ◆ Scalability: Rapidly growing data volume
- ◆ Connectivity: Distributed data sharing

- ◆ 24/7 Availability: No single point of failure
- ◆ High Performance
- ◆ Easy Management



10

- ◆ Virtualisation simplifies the management of storage resources

## Benefits

- ◆ Hides physical storage from applications on host systems
- ◆ Presents a simplified, logical, view of storage resources to the applications
- ◆ Allows the application to reference the storage resource by its *common name*
- ◆ Actual storage could be on complex, multilayered, multipath, storage networks
- ◆ RAID is an early example of storage virtualisation

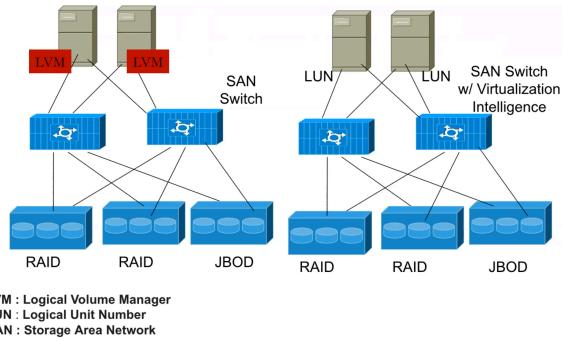
## Types

### Host-Based:

- ◆ On the host
- ◆ Through Logical Volume Manager (LVM)
- ◆ Provides a logical view of physical storage on the host

### Switch-Based:

- ◆ Implemented on the SAN switches
- ◆ Each server is assigned a Logical Unit Number (LUN) to access storage resources
- ◆ Newer powerful switches can operate on the data themselves
- ◆ **Pros:**
  - ◆ Ease of configuration
  - ◆ Ease of management
  - ◆ Redundancy/high availability
- ◆ **Cons:**
  - ◆ Potential performance bottleneck on switch
  - ◆ Higher cost



- ◆ Similar in *topology*
- ◆ Question of where the logic is implemented

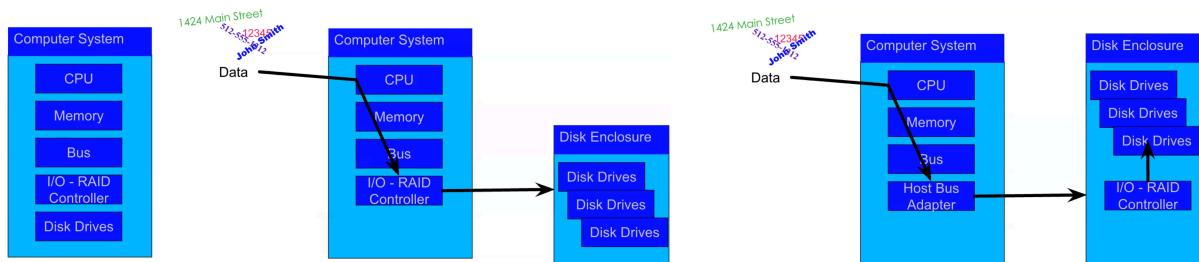
## DAS: Direct Access Storage

---

- ◆ Directly attached to the host
- ◆ No smarts
- ◆ Manual management
- ◆ Good enough, but not for *large scale* systems

### What

- ◆ Simplest scenario: one host
- ◆ RAID controller
  - ◆ Smarts
- ◆ Host Bus Adapter
  - ◆ Not smart
  - ◆ Translate to RAID controller



## SAN: Storage Area Networks

---

- ◆ Deliver content
- ◆ Server access, not client access

### What

- ◆ Specialised high speed network
- ◆ *Large* amount of storage
- ◆ Separate storage from processing
- ◆ High capacity, availability, scalability, ease of configuration, ease of reconfiguration
- ◆ Can be *fibre based*, but now also other channels

### Fibre Channel

- ◆ Underlying architecture of SAN

- ◆ Needs specialised hardware
- ◆ Serial interface
- ◆ Data transferred across *single piece* of medium at *high speeds*
- ◆ No complex signaling
- ◆ SCSI over Fibre Channel (FCP)
- ◆ Immediate OS support
- ◆ *Hot pluggable*: devices can be added or removed at will with no ill effects
- ◆ Provides data link layer above physical layer; analogous to Ethernet
- ◆ Sophisticated error detection at frame level
- ◆ Data is checked and resent if necessary
- ◆ Up to 127 devices (vs 15 for SCSI)
- ◆ Up to 10 km of cabling (vs 3-15 ft for SCSI)
- ◆ **Combines:**
  - ◆ Networks (large address space, scalability)
  - ◆ IO channels (high speed, low latency, hardware error detection)

#### **Original:**

- ◆ Originally developed for fibre optics
- ◆ Copper cabling support: not renamed

#### **Layers:**

1. 0-2: Physical
  - ◆ Carry physical attributes of network and transport
  - ◆ Created by higher level protocols; SCSI, TCP/IP, FICON, etc.

#### **Hardware:**

- ◆ Switches
- ◆ Host Bus Adapters (HBA)
- ◆ RAID controllers
- ◆ Cables

#### **Benefits**

##### **Scalability:**

- ◆ Fibre Channel networks allow for number of nodes to increase without loss of performance
- ◆ Simply add more switches to grow capacity

##### **High Performance:**

- ◆ Fibre Channel fabrics provide a switched 100 MB/s full duplex interconnect

##### **Storage Management:**

- ◆ SAN-attached storage allows uniform management of storage resources

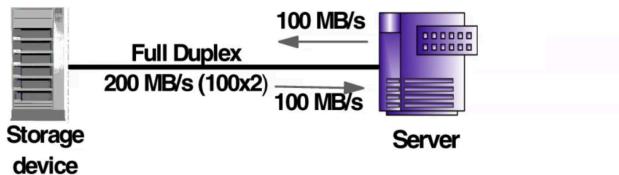
##### **Decoupling Servers and Storage:**

- ◆ Servers can be added or removed without affecting storage
- ◆ Servers can be upgraded without affecting storage
- ◆ Storage can be upgraded without affecting servers
- ◆ Maintenance can be performed without affecting the other part

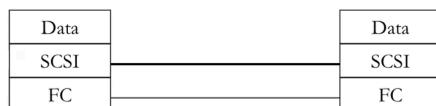
#### **SAN Topologies**

- ◆ Fibre Channel:
  - ◆ Point-to-point
  - ◆ Arbitrated loop; shared medium
  - ◆ Switched fabric

### Point-to-point:

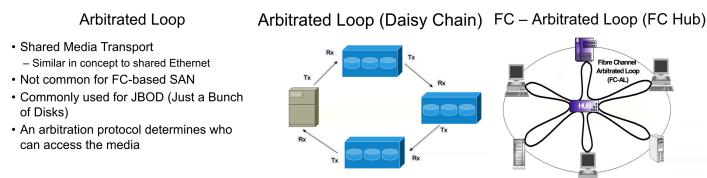


- The point-to-point topology is the easiest Fibre Channel configuration to implement, and it is also the easiest to administer.
- The distance between nodes can be up to 10 km



32

### Arbitrated loop:

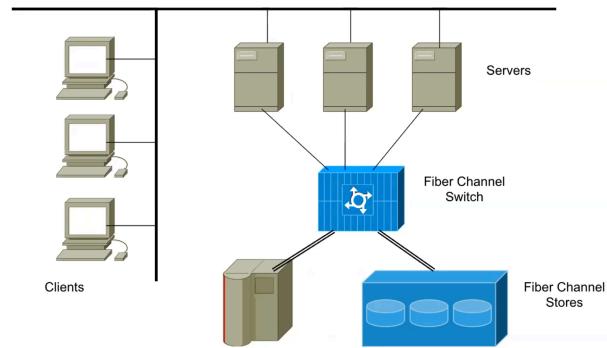


### Switched fabric:

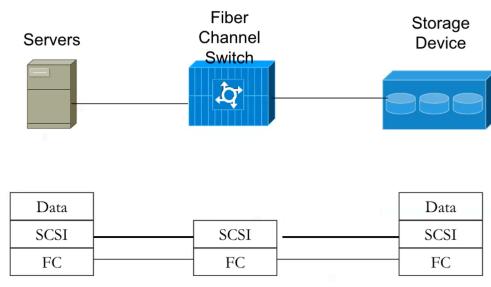
- ◆ This is the most common and “good” one
- ◆ Ways to combine with NAS

## Switched FC SAN

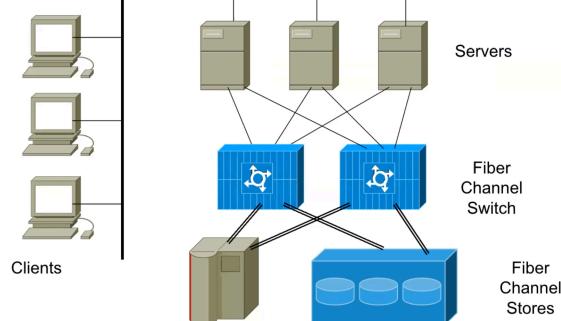
- Fibre Channel-switches function similar to traditional network switches
- Switches provide increased bandwidth, scalable performance, an increased # of devices, increased redundancy
- FC-switches vary in the number of ports and media types they support
- Multiple FC switches can be connected to form a *switch fabric* capable of supporting a large number of host servers and storage subsystems



## Data Access over Switched SAN

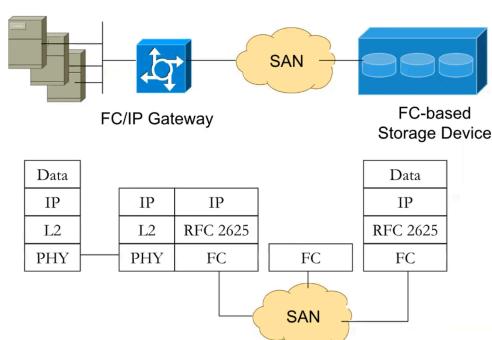


## FC - Storage Area Network (redundant architecture)



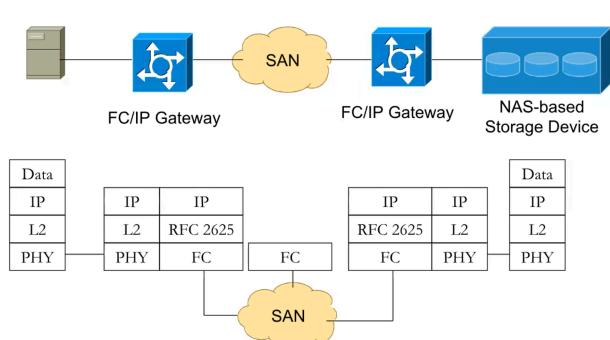
## IP over FC (RFC 2625)

App-1: accessing SAN from IP-based servers



## IP over FC (RFC 2625)

App-2: interworking SAN & NAS



## NAS: Network Attached Storage

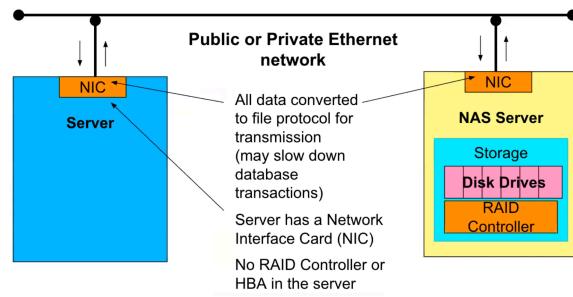
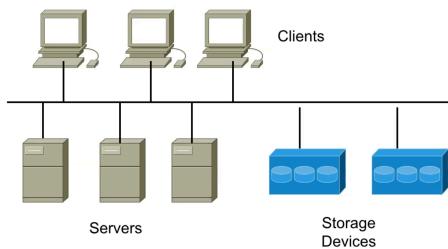
- ◆ A lot of files with different users
- ◆ NFS, AFS fall here

### What

- ◆ A dedicated storage device/server
- ◆ Operates in a client/server mode
- ◆ NAS is connected to the file server via LAN

## Network Attached Storage (NAS)

- Specialized storage device or group of storage devices providing centralized fault-tolerant data storage for a network
- Utilizes a TCP/IP network to "share" data
- Storage "Appliances" utilize a stripped-down OS that optimizes file protocol performance



### Protocol:

- ◆ NFS: Network File System - UNIX/Linux
- ◆ CIFS: Common Internet File System - Windows
  - ◆ Mounted on local system as a drive
- ◆ SAMBA: SMB server for UNIX/Linux
  - ◆ Essentially makes a Linux box a Windows file server

### Drawbacks:

- ◆ Slow speed
- ◆ High latency

### Benefits:

- ◆ **Scalability:** Good
- ◆ **Availability:** Good - Predicated on the LAN and NAS device working
- ◆ **Performance:** Poor - Limited by speed of LAN, traffic conflicts, inefficient protocols
- ◆ **Management:** Decent - Centralised management of storage resources
- ◆ **Connection:** Homogeneous vs Heterogeneous - Can be either

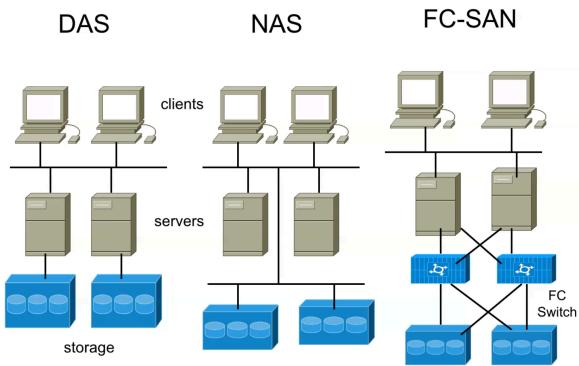
### Alternative Form: iSCSI

- ◆ Internet Small Computer System Interface
- ◆ Alternate form of NAS
- ◆ Uses TCP/IP to connect to storage device
- ◆ Encapsulates native SCSI commands in TCP/IP packets
- ◆ Windows 2003 Server and Linux
- ◆ TCP/IP Offload Engine (TOE) on NICs to speed up packet encapsulation
- ◆ Cisco and IBM co-authored the standard
- ◆ Maintained by IETF
  - ◆ IP Storage (IPS) Working Group
  - ◆ RFC 3720

### Comparison

---

	<b>DAS</b>	<b>NAS</b>	<b>SAN</b>
<b>Storage Type</b>	sectors	shared files	blocks
<b>Data Transmission</b>	IDE/SCSI	TCP/IP, Ethernet	Fibre Channel
<b>Access Mode</b>	clients or servers	clients or servers	servers
<b>Capacity (bytes)</b>	$10^9$	$10^9 \cdot 10^{12}$	$> 10^{12}$
<b>Complexity</b>	Easy	Moderate	Difficult
<b>Management Cost (per GB)</b>	High	Moderate	Low



- ◆ FC: Fibre Channel

## NAS vs SAN

### Traditionally:

- ◆ NAS is used for *low volume* access to large amount of storage by *many users*
- ◆ SAN solution for *high volume* access to large amount of storage, serving data as streaming (typically media like audio/video)
  - ◆ Users are not editing or modifying the data

### Blurred Lines:

- ◆ Kinda need both
- ◆ Complement each other

## **10: Object Storage**

## 11: Compression & Delta Encoding

## 12: Data Deduplication

## 13: Fog/Edge Storage

## **14: Security in Storage Systems**