Distributed Storage Exam Notes

Contents

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 helpt, 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

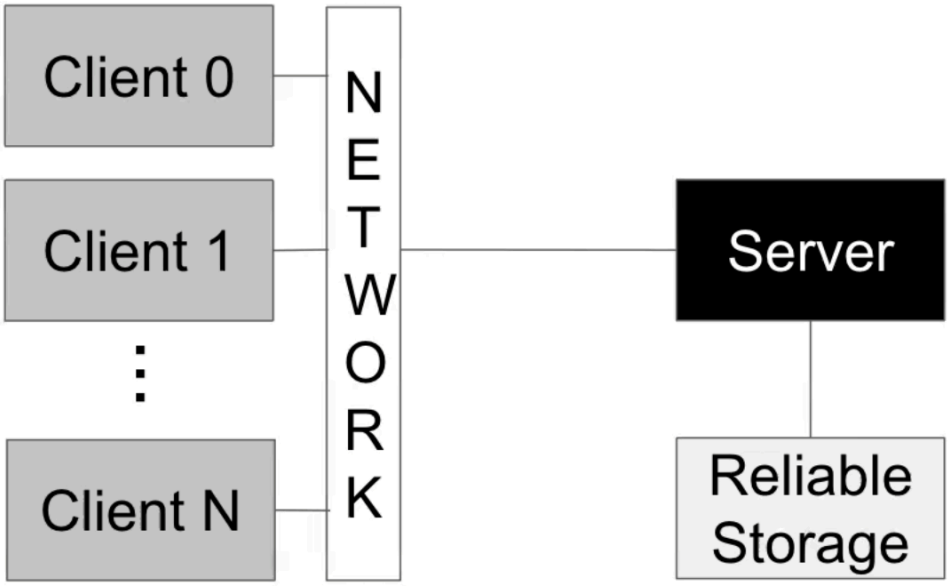
- Get data quickly
- Analyse quickly

> **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 os 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* acces (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 remov
returns: nothing

Removes file name

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

Create new directory name in a directory

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

Remove directory

NFSPROC_READDIR
expects: directory handle, count of bytes to r
returns: directory entries, cookie (to get mor

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

NFSPROC_STATFS
expects: file handle
returns: file system stats
```

Used for pinging the server

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

More than the above.

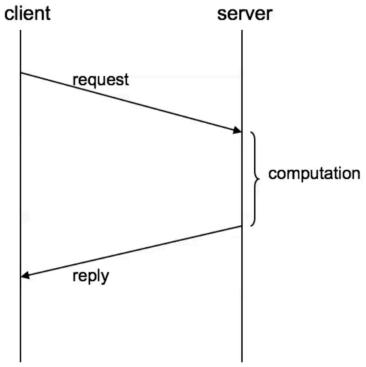
- **`MOUNT`:** protocol takes a directory pathname and returns an `fhandle`
- UNIX-style permission checks, where this information goes through the RPC
  - Root-acces on client give root access on server? NO smh

Idempotency

Property of certain poperations 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 more something like *NFSv4.1*

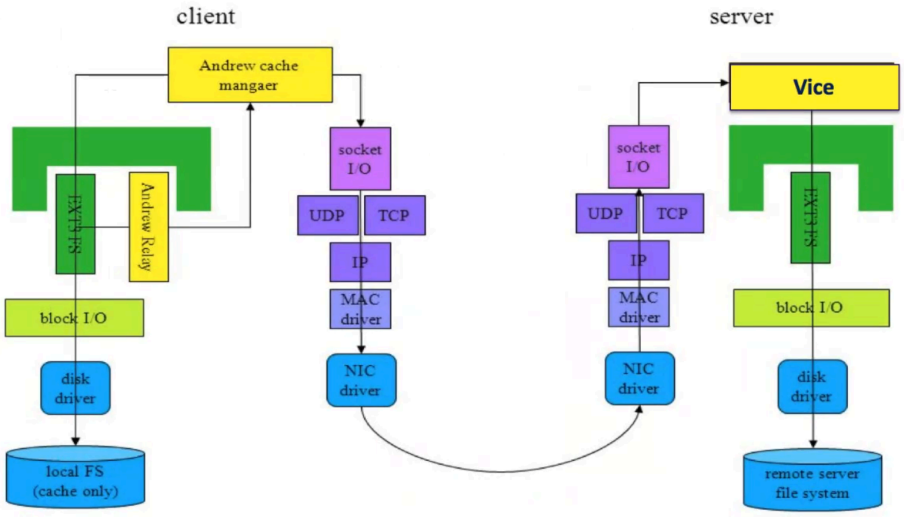
Server: Vice
Client: Venus

Main Goal

Scalability

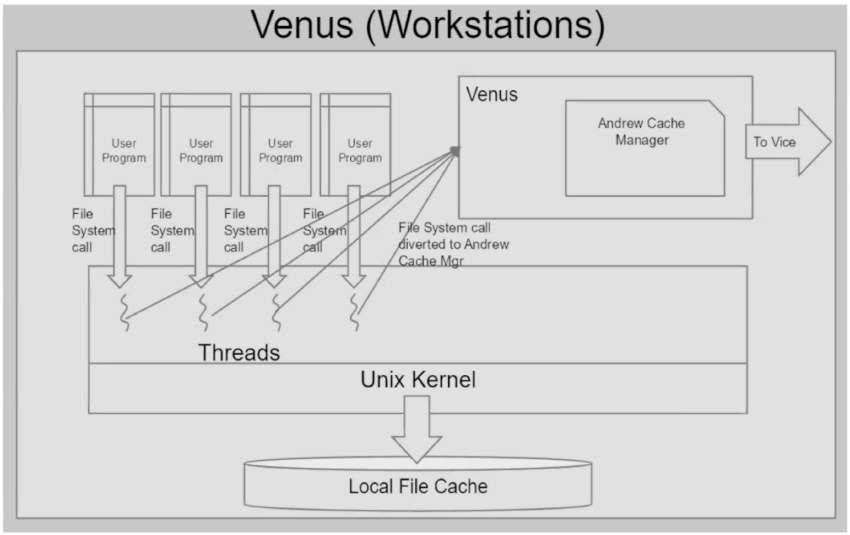
Properties

- 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



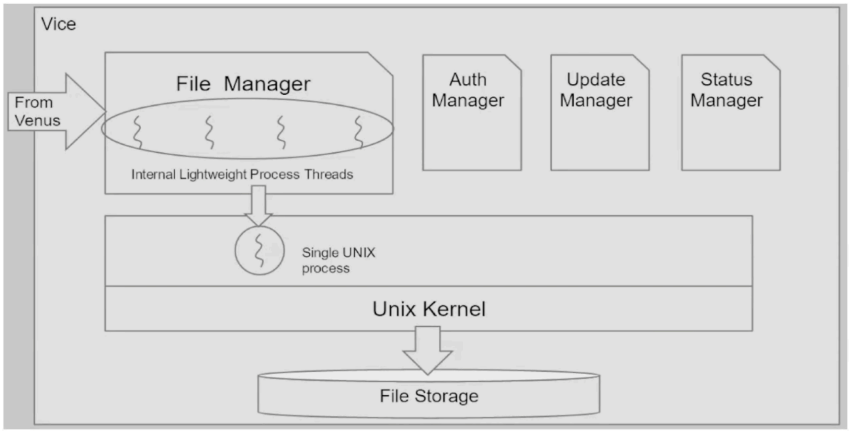
Note that the local file system has to exist and be engaged.

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 → incread 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 ⚠

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)

Client$_1$ P_1	Client$_1$ P_2	Client$_1$ Cache	Client$_2$ P_3	Client$_2$ 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	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

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

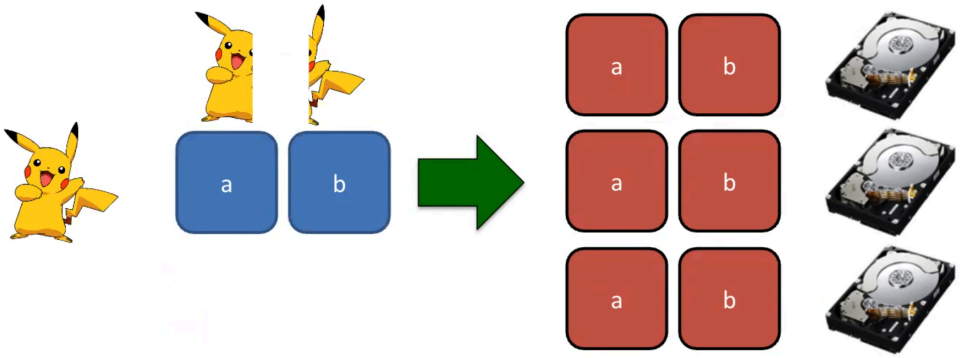
Reliable Storage: Single Server

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



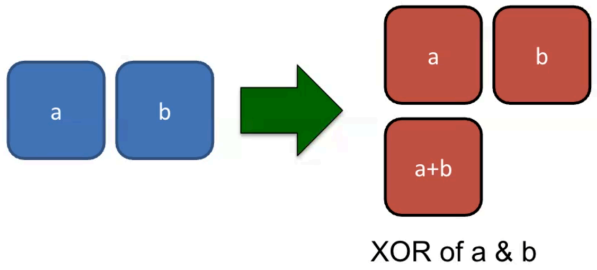
Overhead: 200%

- 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.

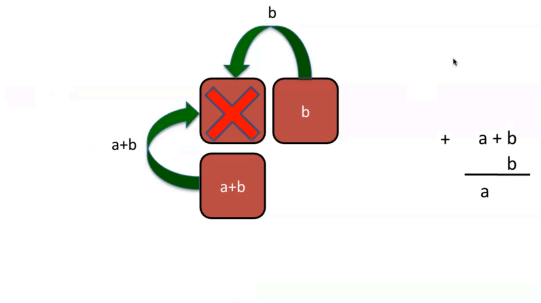
Traffic to repair: 1 unit, 50% of file size

Different Options: Vanilla RAID5



Overhead: 50%

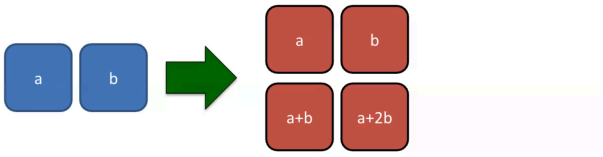
- Can lose *any **one** of the shown fragments*, and still recover it.
- Need to move *the equivalent* of the entire file, no matter how many slices.



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.



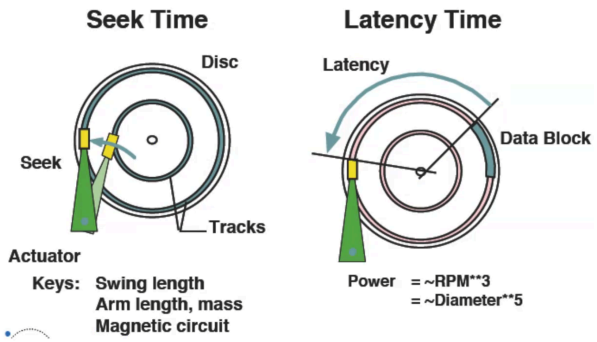
Overhead:	100%
Protection:	2 losses
Traffic to repair loss:	2 units (100% of file size)

04: RAID systems

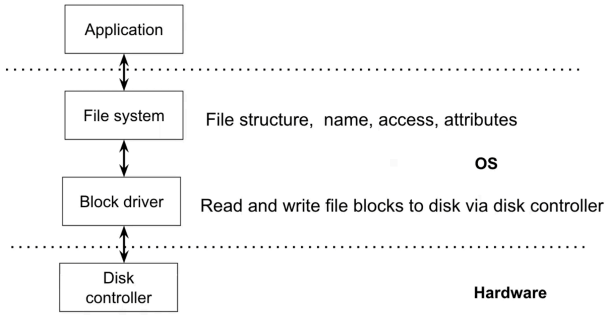
Proporties

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

Disks



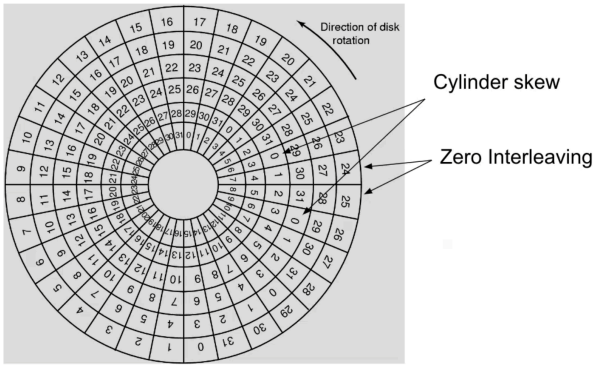
$$\text{access time} = \text{seek time} + \text{latency time} + \text{transfer time} \tag{1}$$



Disk Arm Scheduling Algorithm

1. Seek time - Move to the right cylinder
2. Latency time - Sectors rotate under the head
3. 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 😯

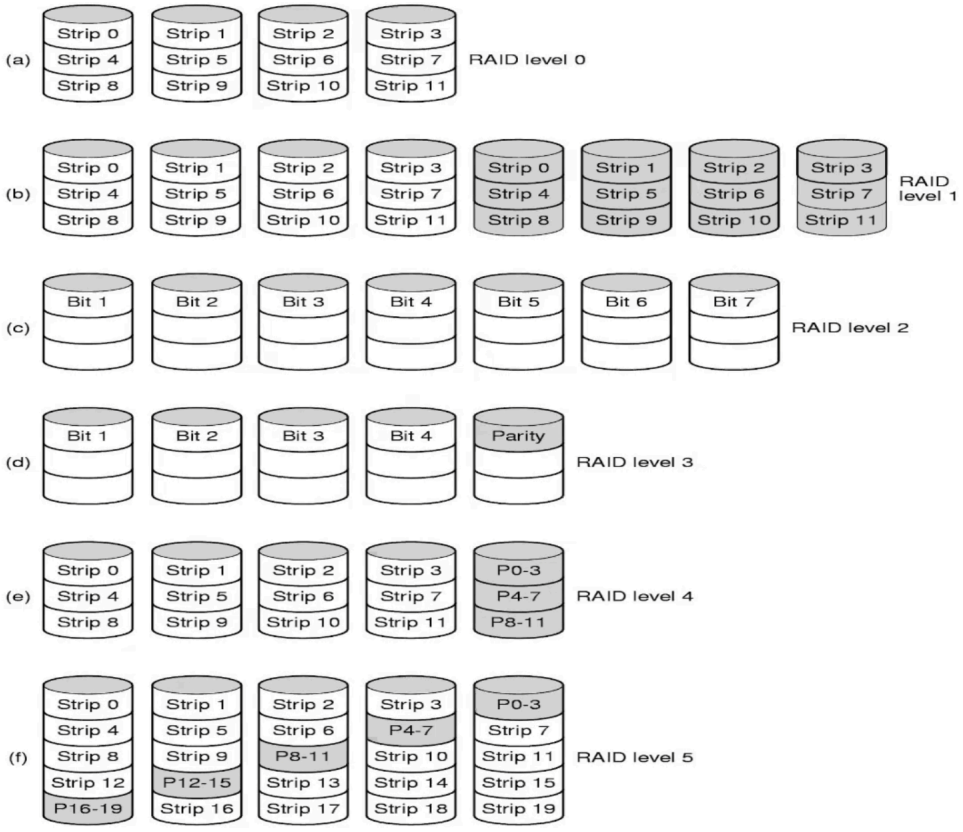
RAID

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.

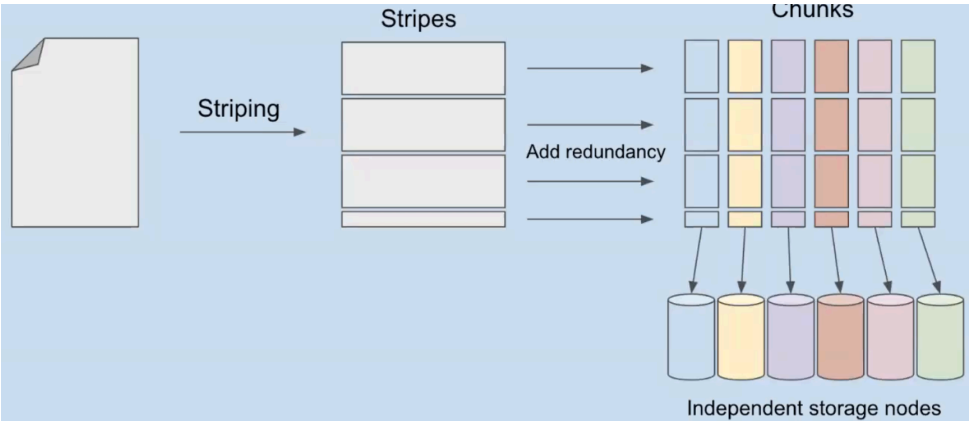
What is RAID

Redundant Array of Inexpensive Disks
Redundant Array of Independent Disks

Alternative to SLED: Single Large Expensive Disk

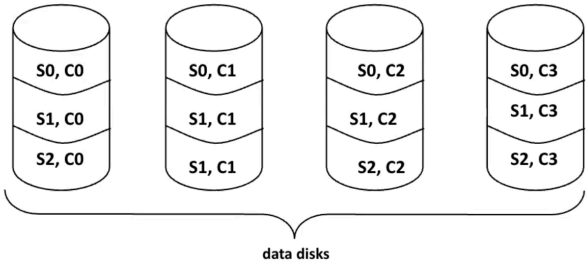
- A RAID box with a RAID controller looks just like a SLED to the system using it

Striping



RAID0

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



`Sx, Cy = Stripe x, 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

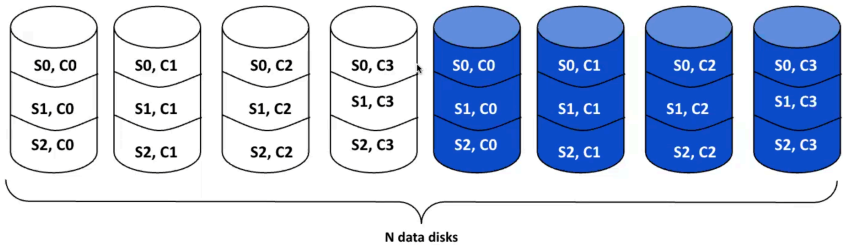
- Meant for *reliability*
- **Key:** N mirrored disks
  - 1 Main disk, $N - 1$ mirrors
  - Full file replication
- **Failure:** Use any surviving disk, plug in new disk, copy over
- **Read:** Pick fastest disk, ~$2x$ for $N = 2$, with a lot of requests, parallel read from several drives for different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

RAID1	
Metric	**Evaluation**
Space Efficiency	$1/N$, very *bad*
Storage Capacity	Capacity of *smallest* 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*



`Sx, Cy = Stripe x, Chunk y`

RAID1 E	
Metric	**Evaluation**
Space Efficiency	$1/2$, typically used with 2 replicas
Storage Capacity	$NC/2$, half capacity of N disks
Fault Tolerance	$F = 2$, can cause loss in worst case, best case: $F = N/2$
Read Performance	$N/2$, light load, up to N under high load
Write Performance	$N/2$ (could be higher depending on placement strategy)
Min Disks Required	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.

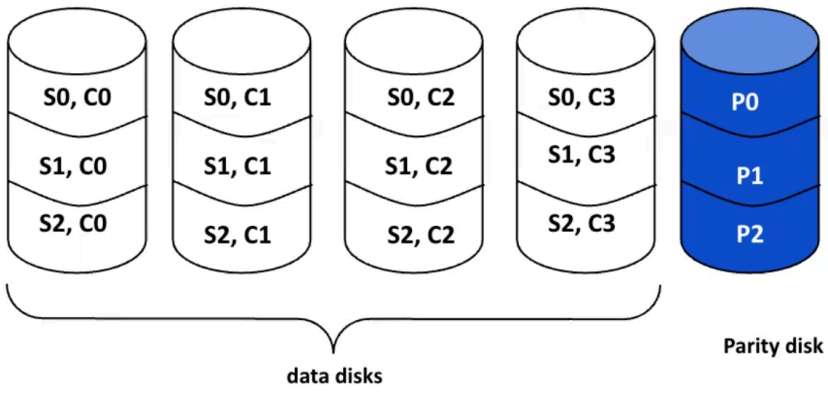
However, it does of course require some parallel setup like this, which I didn't have.

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*

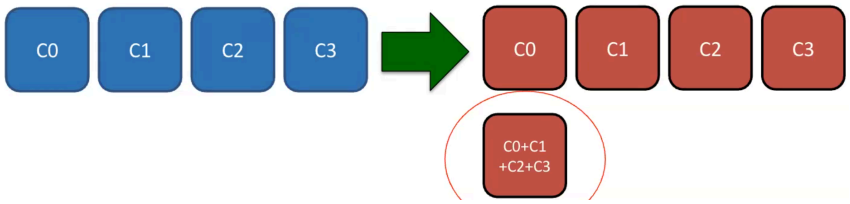
Like `RAID0` but with parity



`Sx, Cy, Pz = Stripe x, Chunk y, 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

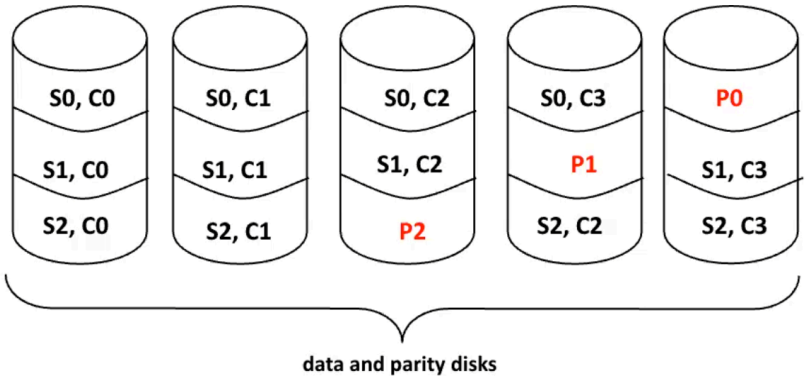
Challenges

- Seperated 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



`Sx, Cy, Pz = Stripe x, Chunk y, Parity z`

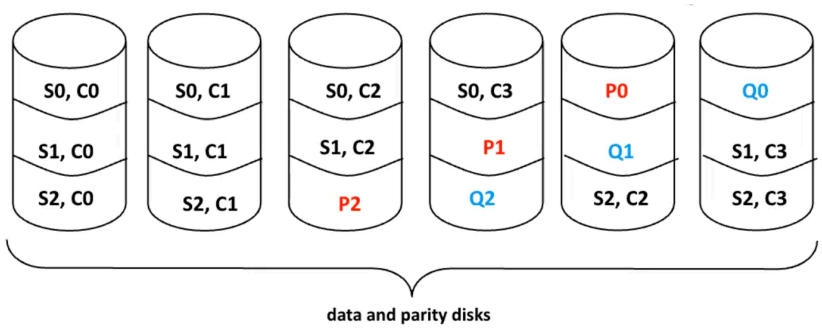
RAID5	
Metric	**Evaluation**

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, as we're still losing the capacity equal to a whole disk on parity
Fault Tolerance	$F = 2$, same as `RAID4`
Read Performance	N, no longer impaired by the parity disk
Write Performance	$N - 1$, we still have to write parity each time, same as `RAID4`
Min Disks Required	3, same as `RAID4`

`RAID6`

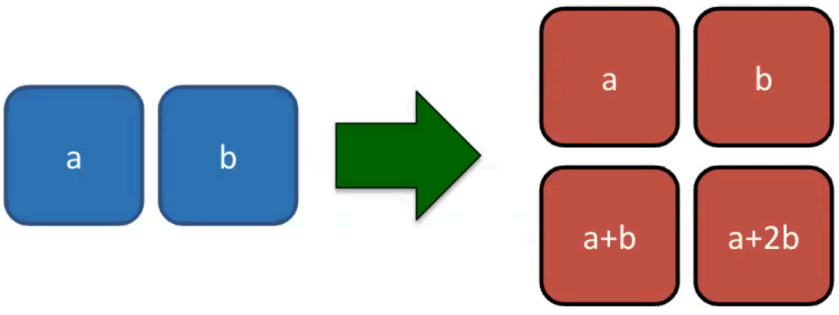
- **Read:** can outperform `RAID5`, as there are multiple parities to take from
- **Write:** slower than `RAID4`, because we need to perform more parity
- Better *reliability* than `RAID4`

Like `RAID5` with an extra parity

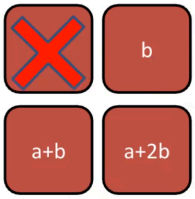


`Sx, Cy, Pz, Qw = Stripe x, Chunk y, Parity z, Parity 2 (Q) w`

- Linear combination of 2 chunks, in such a way that you dont 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

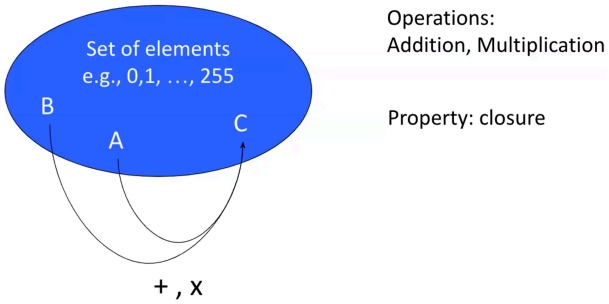
RAID6	
Metric	**Evaluation**
Space Efficiency	$1 - 2/N$, same as `RAID5` but taking the space of another parity disk into account
Storage Capacity	$(N-2) \times C$, for N disks of same capacity C
Fault Tolerance	$F = 3$ because of the extra parity
Read Performance	N, same as `RAID5`, but can sometimes 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 $\mathrm{GF}(2^2)$: 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 $\mathrm{GF}(2^2)$: 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

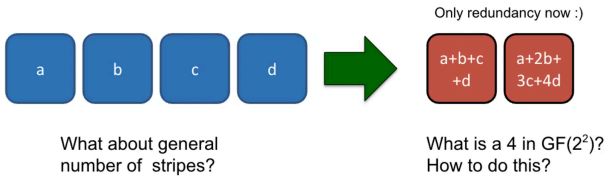
Using the two tables above

```
a = 10
b = 11

2 * b = 10 * 11 (= 2 * 3)
      = 01 (= 1)

a + 2 * b = 10 + 01
          = 11
```

- *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`

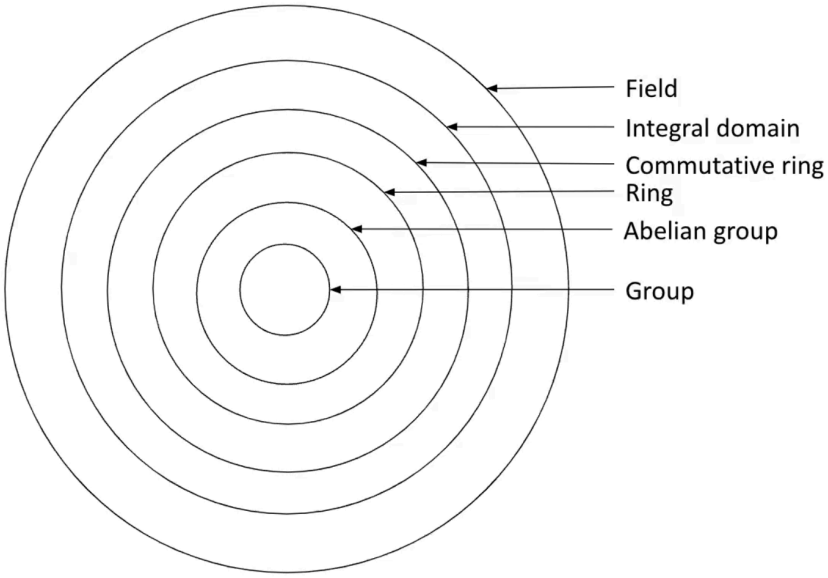
- Combine several `RAID` levels stacked
- Combines the different properties
- Typically uses two levels like `RAID 10`
  - **First level is the lowest:** `RAID1` replication at the bottom
  - **Second is the highest:** Data striping of `RAID0` (carried out first) on top of `RAID1` replication

*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, +, *x*} 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} . a) ≡ 1

2.- Can use the extended Euclidean algorithm

- $\mathrm{GF}(2^n)$ 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 1×1 or 3×3, similar case with 3.

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

Ordinary polynomial arithmetic:

A polynomial of degree n

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

a_i are the coefficients, chosen from a set

Operations:

Addition $f(x) + g(x) = \Sigma (a_i + b_i) x^i$

Multiplication $f(x) \times g(x) = \Sigma C_i x^i$

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

- 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., $(a_i + b_i)$ mod 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 mod p(x) operation: divide by p(x), keep the remainder
  - Divide by some irreducible polynomial (of order n?)
  - How about 2 +3?

    $2 = (10)_b$ and $3 = (11)_b$

    As polynomials:

    $2 \equiv x$ 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 $\mathrm{GF}(2^8)$, where you might *need* to compute

How to implement multiplication 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) = 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 = $(00010001)_b$ and m = $(10100111)_b$

```
   101001110000      (m shifted 4 times)
(XOR)  10100111      (m shifted 0 times)
-------------------
   101011010111
```

B.- Modulo irreducible polynomial

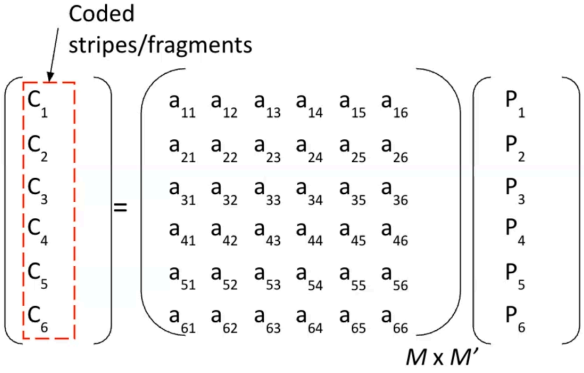
p(x) = $x^8 + x^4 + x^3 + x + 1$ $(100011011)_b$

```
   101011010111   mod  p(x)
(XOR)100011011000
-------------------
   001000001111
(XOR)001000110110
-------------------
   000000111001   → (00111001)b
```

Linear Coding

- Generating a linear coded fragment/stripe C

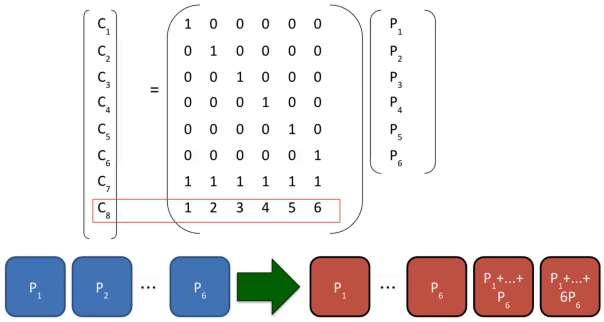
$$C_i = \sum a_{ij} P_j \tag{2}$$



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

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

RAID6



How to Decode (RAID6)

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

Example:

- Lost 5 and 6

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

Something something linear system of equations

→ *qaussian 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
$0 \rightarrow 0$
$1 \rightarrow 1$
$2 \rightarrow x$
$3 \rightarrow x+1$
$4 \rightarrow x^2$
$5 \rightarrow x^2+1$
$6 \rightarrow x^2+x$
$7 \rightarrow x^2+x+1$

Mapping to polynomial
$2^2 \rightarrow x^2 \rightarrow 4$
$2^3 \rightarrow x^3 \rightarrow R(x^3/x^3+x^2+1) \rightarrow x^2+1 \rightarrow 5$
$2^4 \rightarrow x^4 \rightarrow x(x^2+1) \rightarrow x^2+x+1 \rightarrow 7$
$2^5 \rightarrow x.x^4 \rightarrow x(x^2+x+1) \rightarrow x+1 \rightarrow 3$
$2^6 \rightarrow x.x^5 \rightarrow x(x+1) \rightarrow x^2+x \rightarrow 6$
$2^7 \rightarrow x.x^6 \rightarrow x(x^2+x) \rightarrow x^3+x^2 \rightarrow 1$

- 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

- Numoerous 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 tupically 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 aabout erasure codes; striping with redundancy, e.g. `RAID`-like, array codes
- This week will be a bout *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 GF(2^n)?

A.- Product (shifts+ XORs)

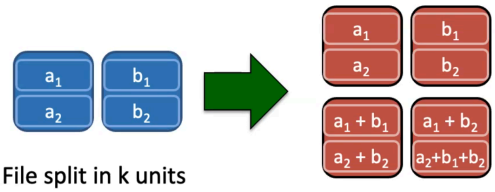
1) Pick one as multiplier (M) and another as multiplicand (m)
2) For each "1" in "M", left shift the "m" by the position of the "1"
3) XOR shifted versions

B.- Modulo irreducible polynomial (long division)

4) Initialize: F(0) = 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

Example

Generate:

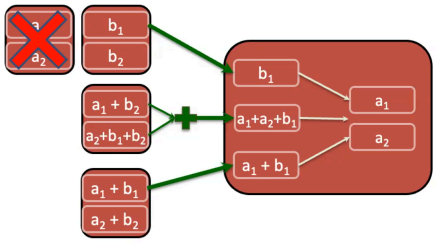


File split in k units

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 beahviour

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

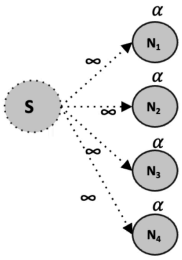
$$\frac{n-1}{n-k}\frac{B}{k} \quad \text{bits}$$

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

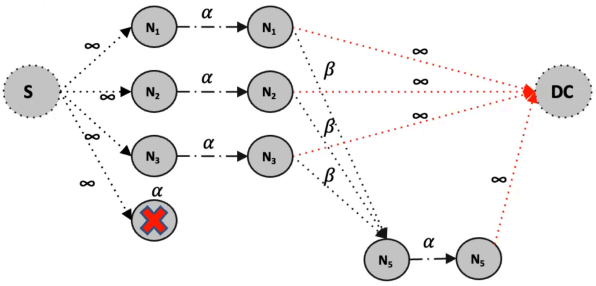
Previous Example: For *n* = 4, *k* = 2, then at least ¾ *B* bits are needed (and we used exactly that)

Graph

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



- Transmit β bits from each surviving node (d surviving nodes)
- Total bandwith 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

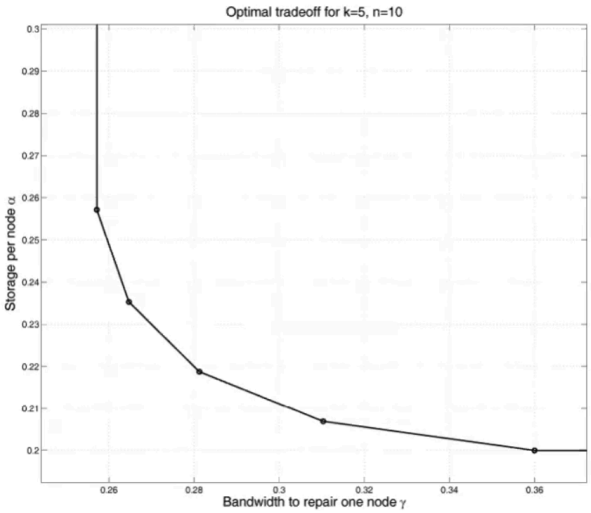
Full theorem:

- Clear connection between the degree d and other parameters

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

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

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



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

Minimum Bandwidth Regenerating (MBR):

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

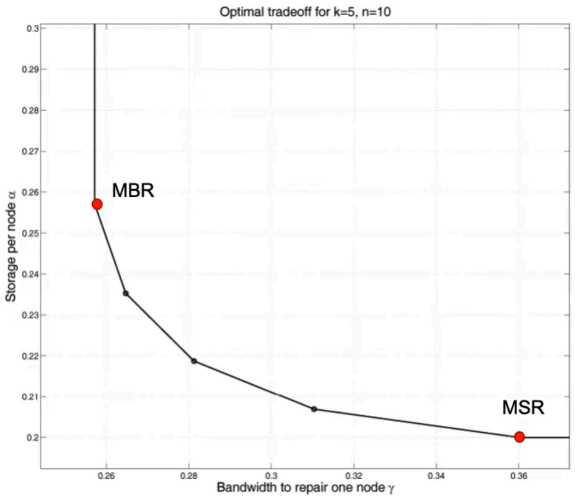
Consider case of d = n – 1

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

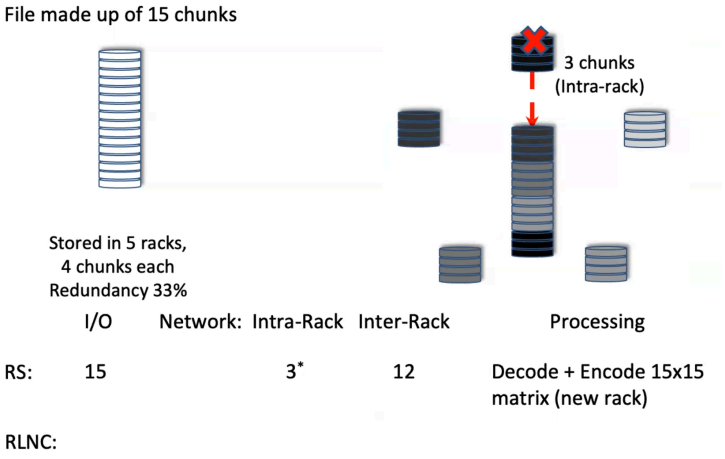
- What happens if we only lose one or several disks on a node, instead of all of it

⚠ Possible Report Weakness

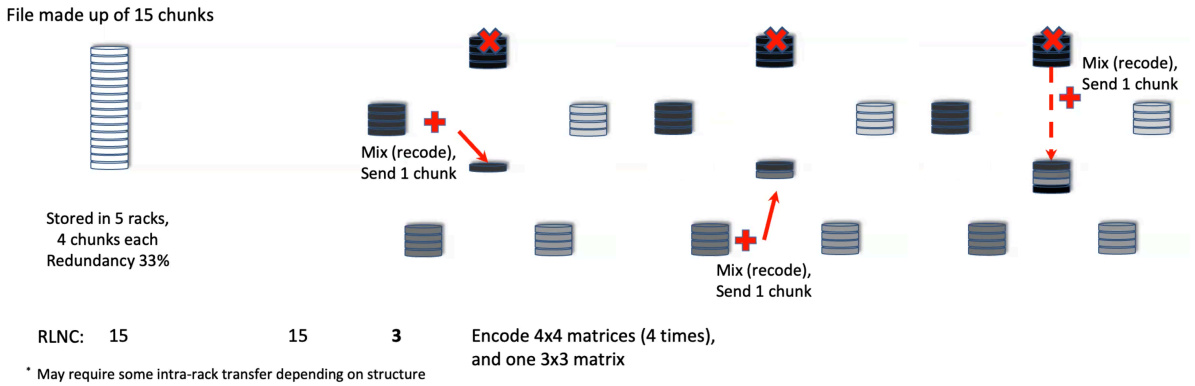
Not a weakness but the report doesn't talk about node vs inter-node disk loss

RS vs RLNC

Example: Reed Solomon
Collect 15 chunks → decode → re-encode
Centralised processing at one node



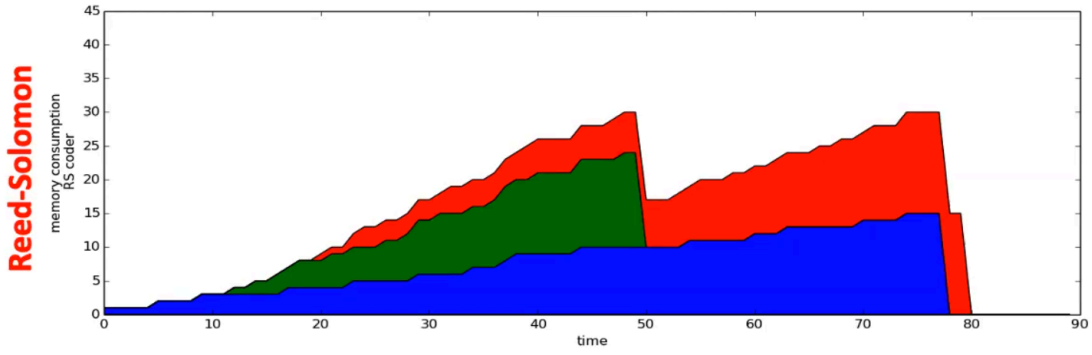
Example: Random Linear Network Co de Distributing the processing between nodes

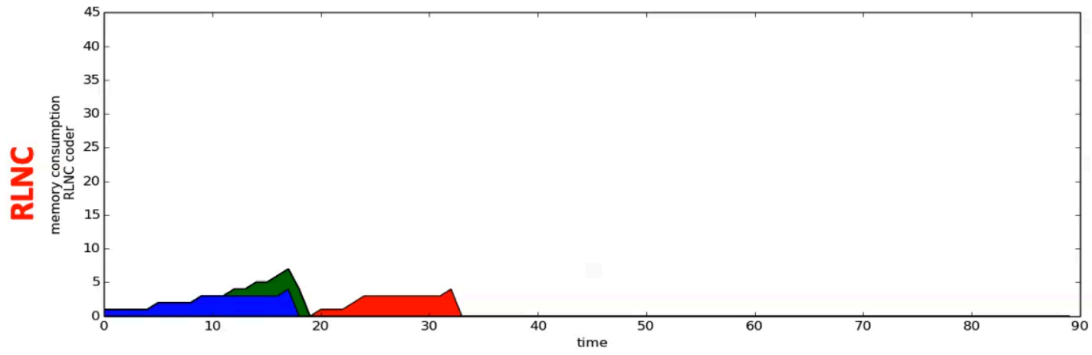


Benefits:

- Less memory consumption furing 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





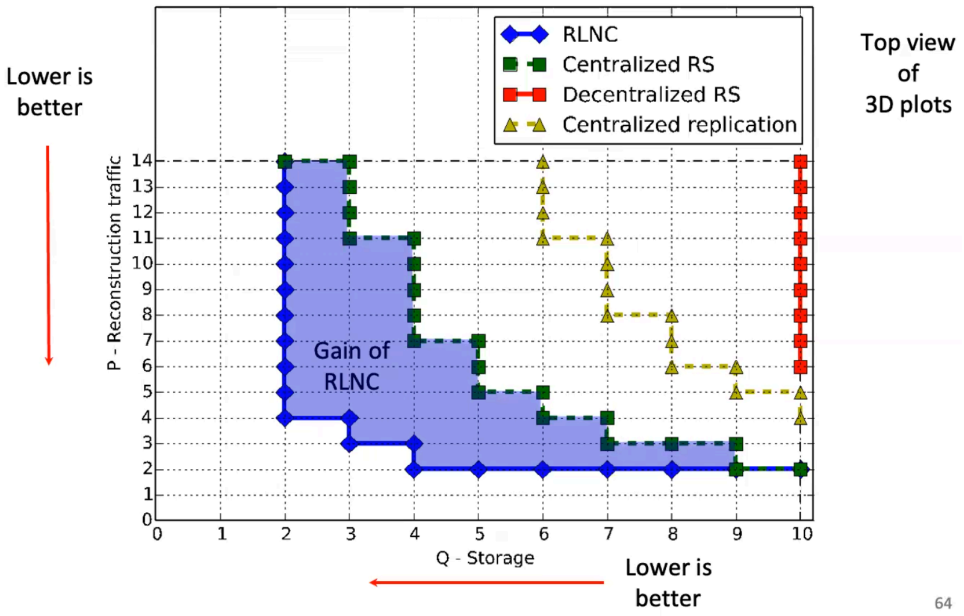
⍰ Possible Report Weakness

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



Exact Repair

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

- Minimise repair bandwidth while maintingin 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 multiplcations of data?)

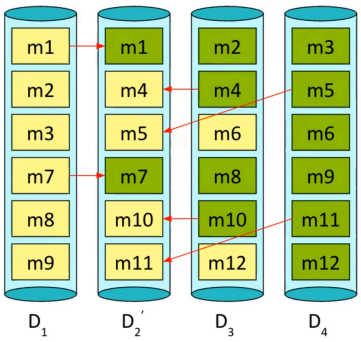
- Regenerating codes (RLNC)

How to do Exact Repair with RLNC

*MBR and MSR points are predicated on **functional repair***

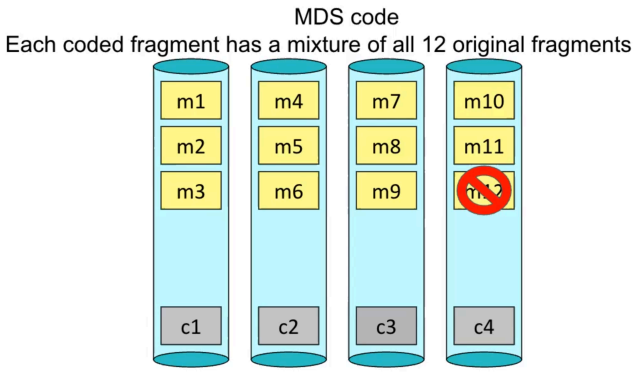
RLNC can still be exact repair, but that restricts it in many ways, making it less efficient.

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



If D_2 had was lost, it can be rapaired easily ↑

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



"Interference Cancellation"
If we know c4 = m1 + 2 x m2 + 3 x m3 + … + 11 x m11 + 12 x m12

Local Repairability

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

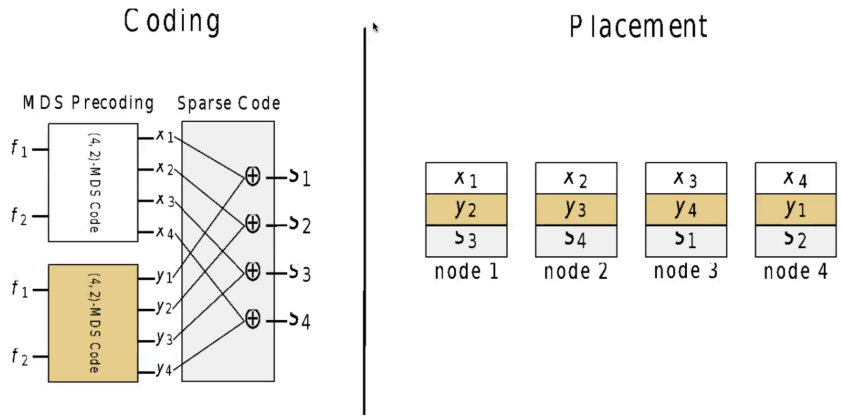
Theorem 1 For locality r, there is a related storage cost

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

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

Example

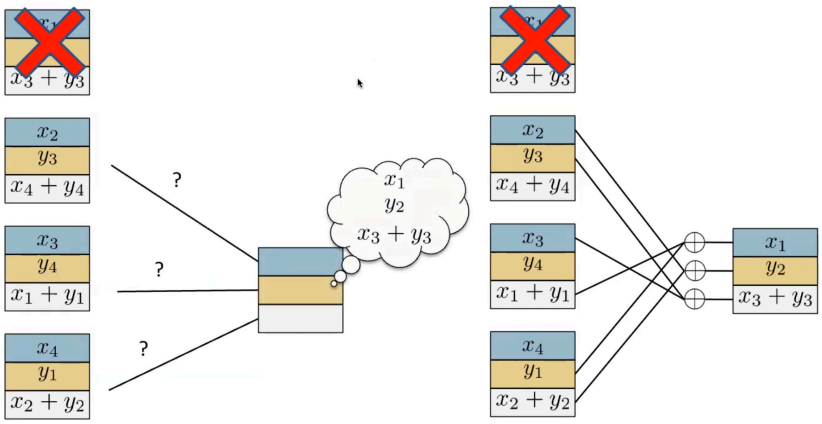
We want to have locality



- 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

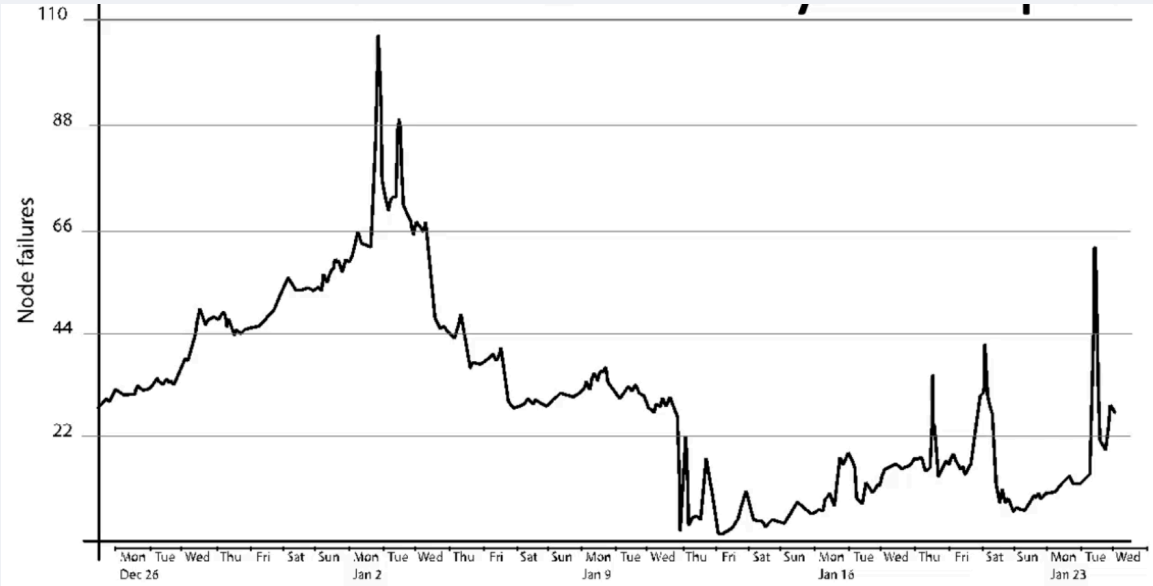
How to repair



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

⚠ Possible Report Weakness

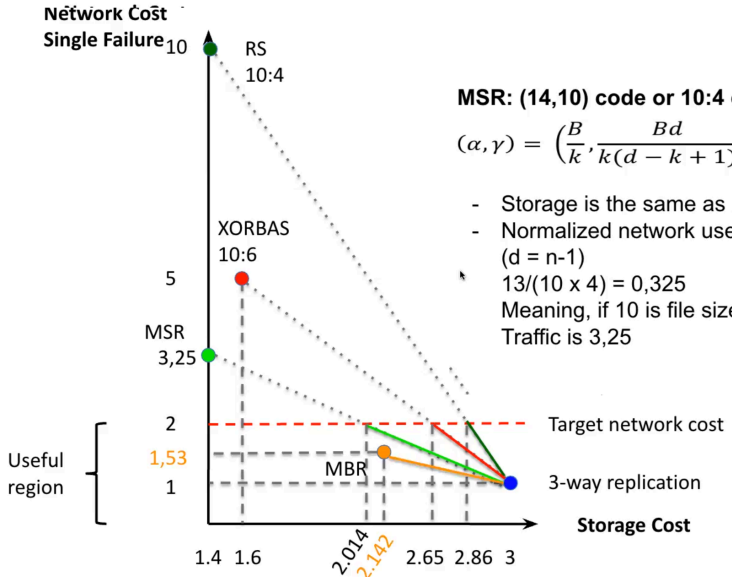
He likes the problem from the Facebook setup.
→ The *big picture* view



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



MSR: (14,10) code or 10:4 code

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

- Storage is the same as RS
- Normalized network use is (d = n-1)
  13/(10 x 4) = 0,325
  Meaning, if 10 is file size, then Traffic is 3,25

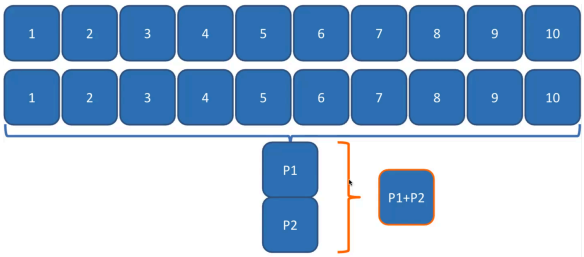
MBR: (14,10) code or 10:4 code

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

- Storage is *not* the same as RS
  14 x α = 2,142
- Normalized network use is (d = n-1)
  13x2/(2x 10 x 13 - 100 +10) = 0,153
  Meaning, if 10 is file size, then Traffic is 1,53

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 ~ 1.13 file
Reliability: Recovery from at least 4 losses
Processing: encoding 1 file when losing P1, P2, or P1+P2

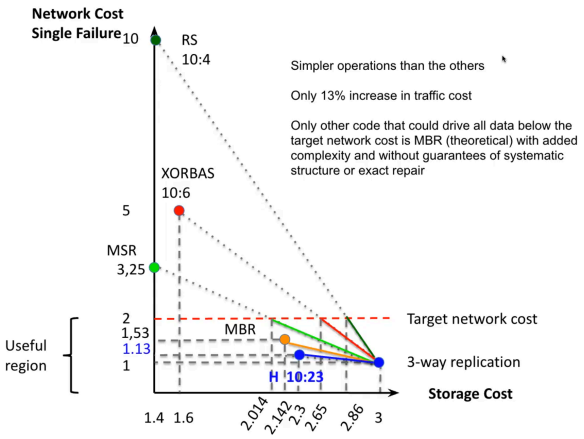
Can this help?

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 mocing the data for the intial 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

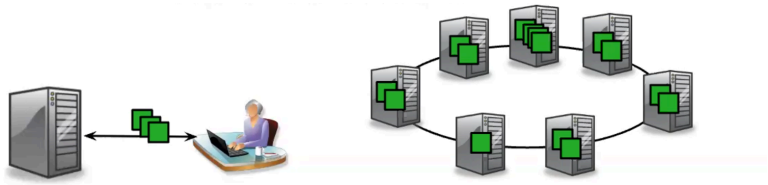
Hadoop File system

- 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 iles
  - 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*

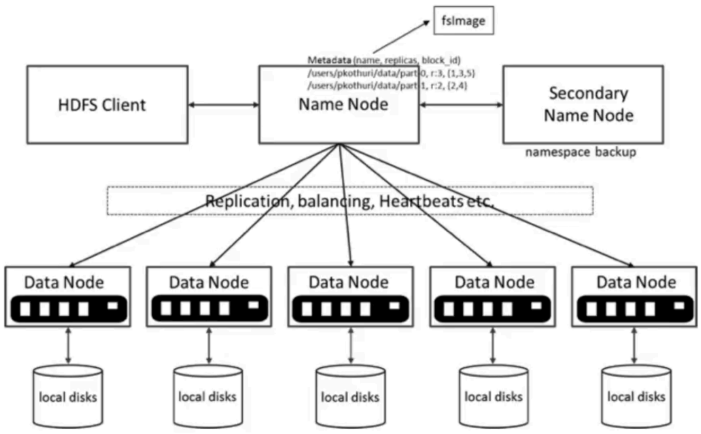
More more

- 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

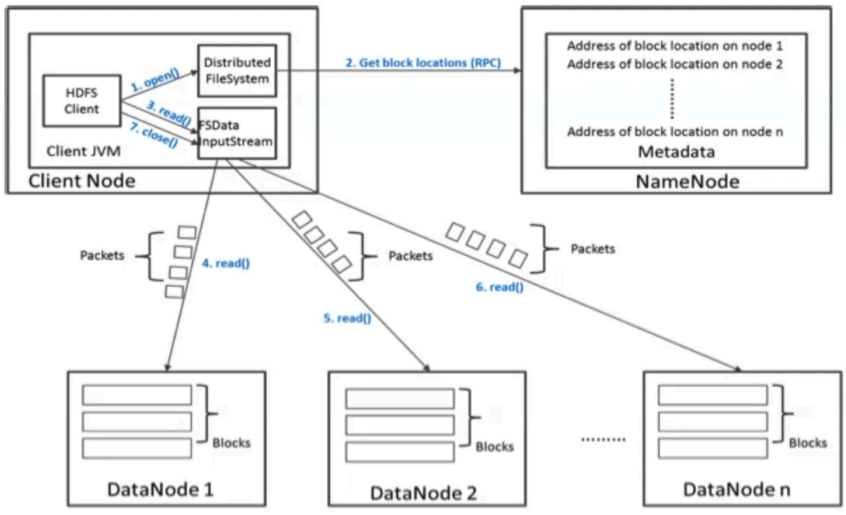
Resiliancy

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:

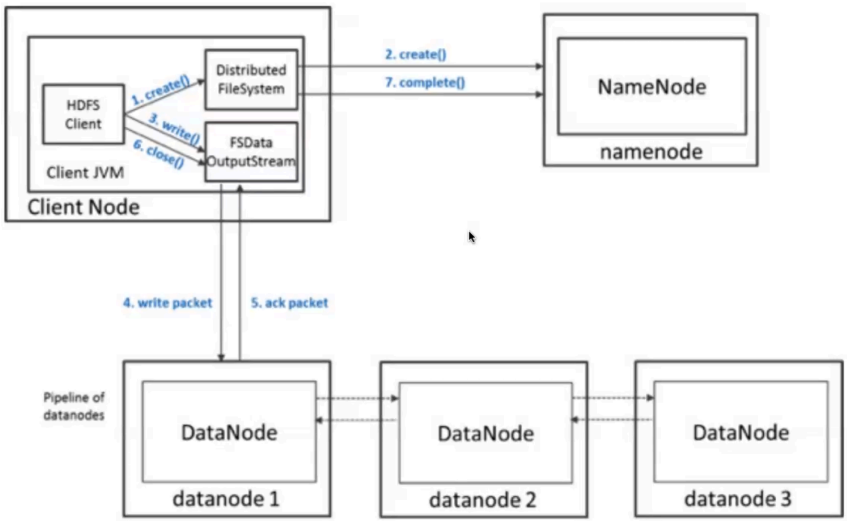
Clearly my client does not simply use the Name/coordinator-node to access metadata and then connect to data/worker-nodes itself.

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
  - Aditionally, data transfer brandwidth is multiplied
- Computation is near data
- Replication factor, default: $k = 3$

Block placement:

- After placement you have a certain structure, and knowdledge 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 heartbets 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:

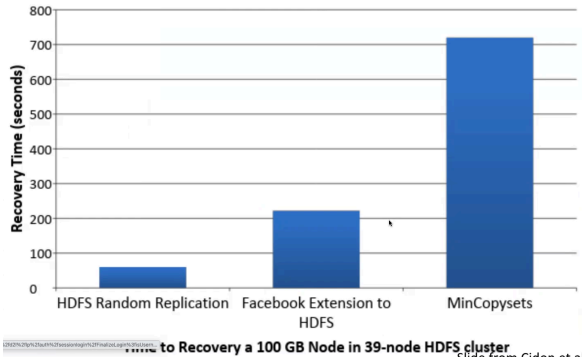
1% of nodes fail concurrently:

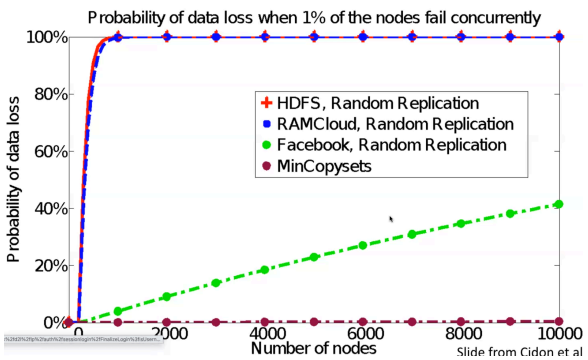
- Chance of data loss doens't mean that percentage of data is lost
- It is a percentage of the probability of data loss, so at least 1 chunk lost

	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:

$$\frac{\binom{F}{R}}{\binom{N}{R}} \tag{3}$$

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

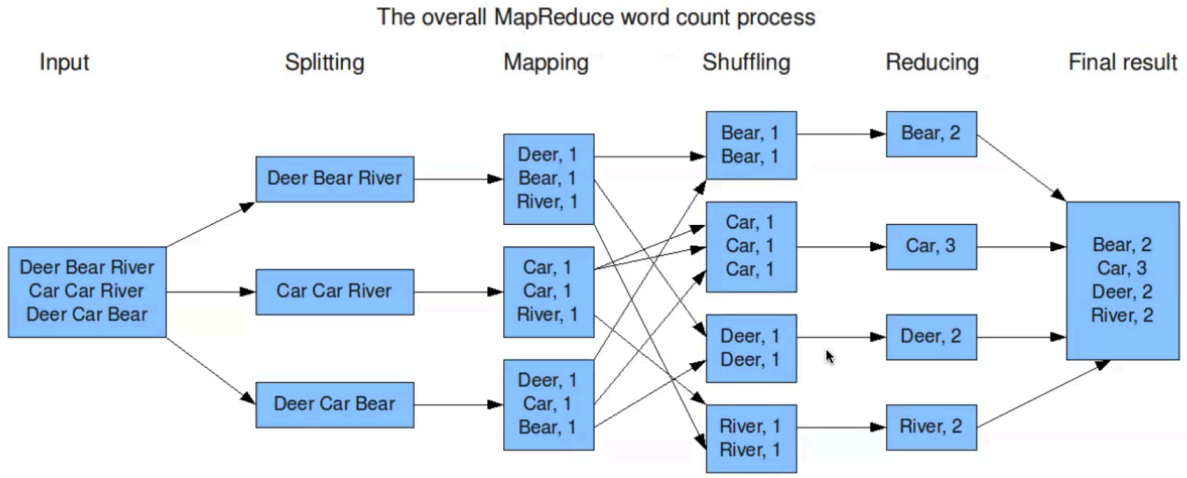
- For every key identify what the value is

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

Example: Count words of same type

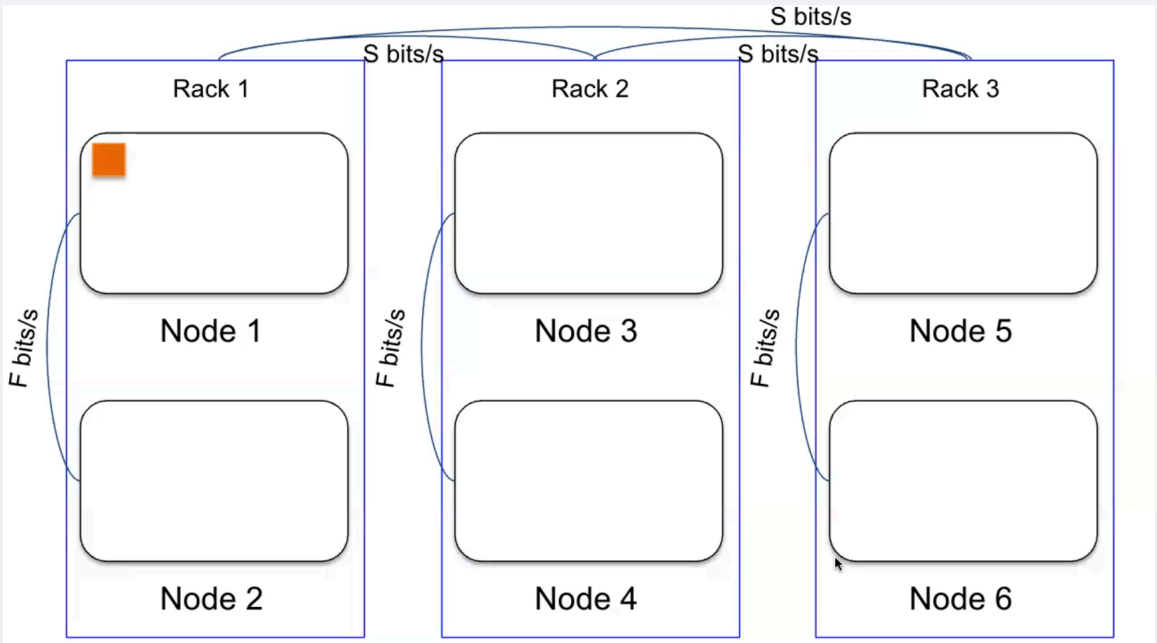


- Already at the splitting stage when stuff has been chunked and split into nodes.
- What the mapping and reduction does is defined base on what function you want to compute.
- In this case the mapping could actually also reduce the data by counting each value up, if a key is repeated.

Steps:
1. **Mapping:** Take each word and write it's value of 1 such that you can count them in reduction stage.
2. **Shuffle:** Move data around such that data for a single key is in a single node.
3. **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:



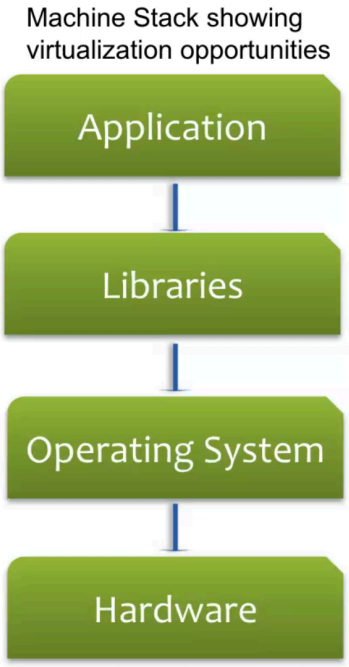
1 - How would Hadoop create 3 replicas in total? Write all set of nodes that could be selected
2 - Time to replicate a file of D bits From Node 1 using of the sets in (1) - Called T1
3 - Considering that replicas need to be generated in Node 3 and Node 5,
Compute the time to replicate (T2)
Compute the Gain of Hadoop w.r.t. this strategy : G = T2/T1
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



- 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

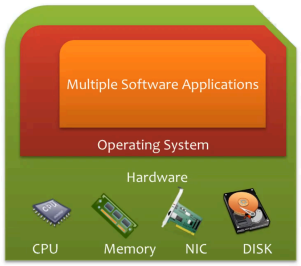
Primary approaches to virtualization

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

Hypervisor

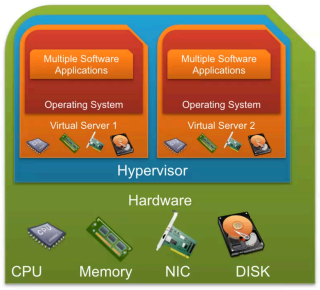
Enables the capability of manipulating and running several OS on the same hardware.

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

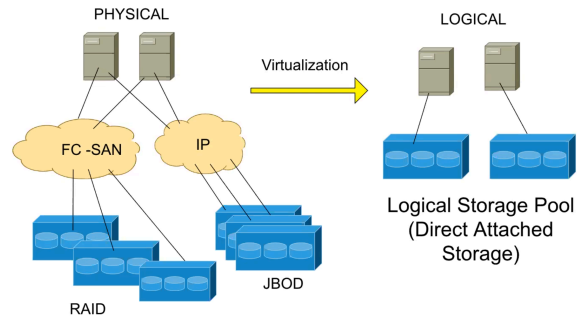


- Can run multiple OS simultaneously
- Each OS can have different hardware configuration
- Efficient utilization of hardware resources
- Each virtual machine is independent
- Save electricity, initial cost to buy servers, space
- Easy to manage and monitor virtual machines centrally

Virtualisation Solves

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

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



JBOD (abbreviated from "**Just a Bunch Of Disks**"/"**Just a Bunch Of Drives**") is an architecture using multiple hard drives exposed as individual devices.

- 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

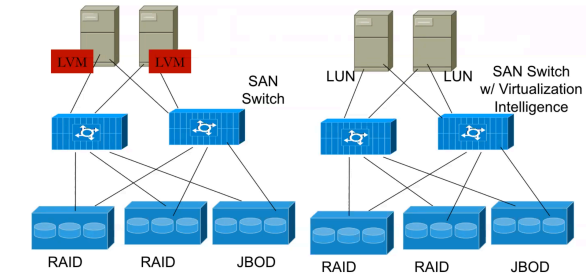
Types

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



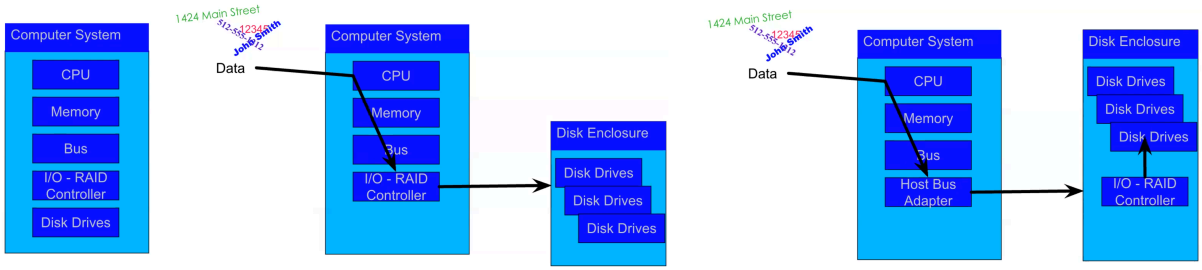
- Similar in *topology*
- Question of where the logic is implemented

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
- Privdes 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

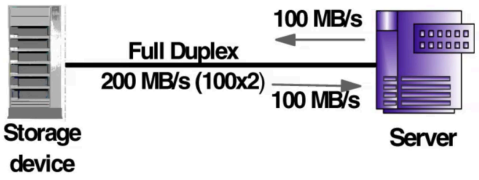
High Performance:
- Fibre Channel fabrics provide a switched 100 MB/s full duplex interconnect

Storage Management:
- SAN-attached storage allows uniform management of storage resources

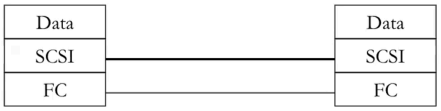
Decoupling Servers and Storage:
- Servers can be added or removed without affecting storage
- Servers can be upgraded without affecting storage
- Storage can be upgraded without affecting servers
- Maintenance can be performed without affecting the other part

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



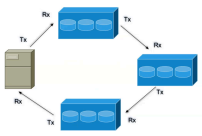
32

Arbitrated loop:

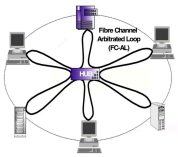
Arbitrated Loop

- Shared Media Transport
  - Similar in concept to shared Ethernet
- Not common for FC-based SAN
- Commonly used for JBOD (Just a Bunch of Disks)
- An arbitration protocol determines who can access the media

Arbitrated Loop (Daisy Chain)



FC – Arbitrated Loop (FC Hub)

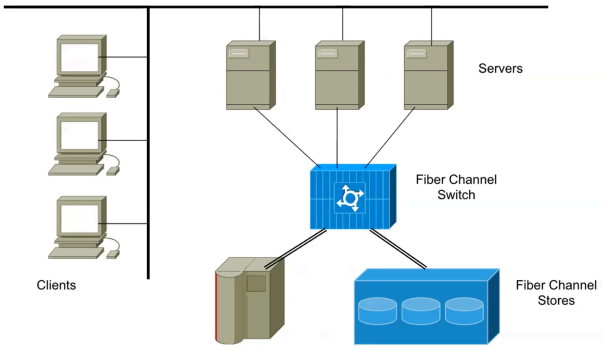


Switched fabric:

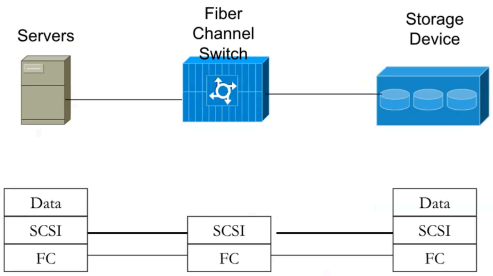
- This is the most common and *"good"* one
- Ways to combine with NAS

Switched FC SAN

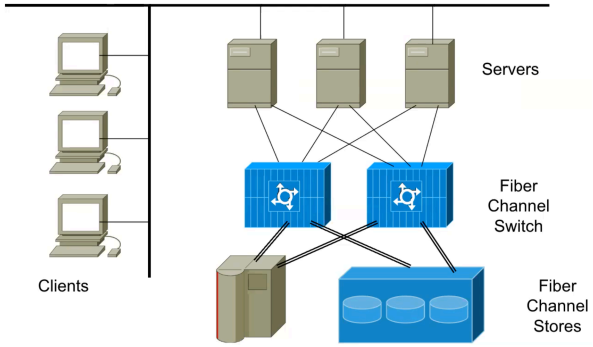
- Fibre Channel-switches function similar to traditional network switches
- Switches provide increased bandwidth, scalable performance, an increased # of devices, increased redundancy
- FC-switches vary in the number of ports and media types they support
- Multiple FC switches can be connected to form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



Data Access over Switched SAN

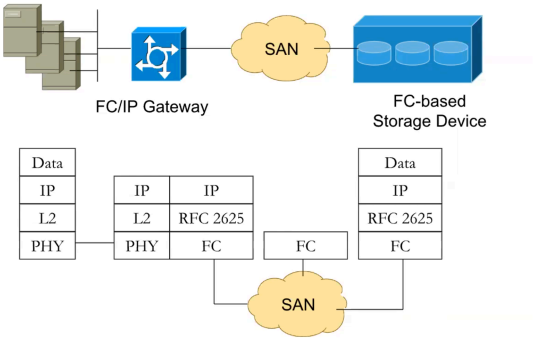


FC - Storage Area Network (redundant architecture)



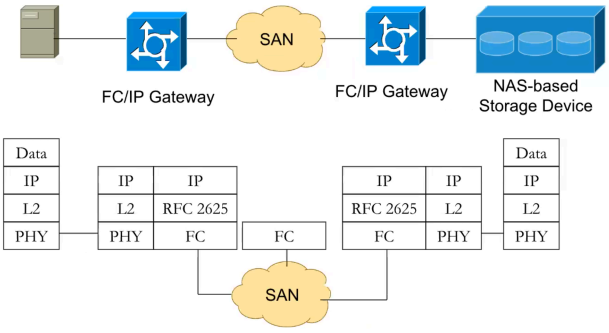
IP over FC (RFC 2625)

App-1: accessing SAN from IP-based servers



IP over FC (RFC 2625)

(App-2: interworking SAN & NAS)

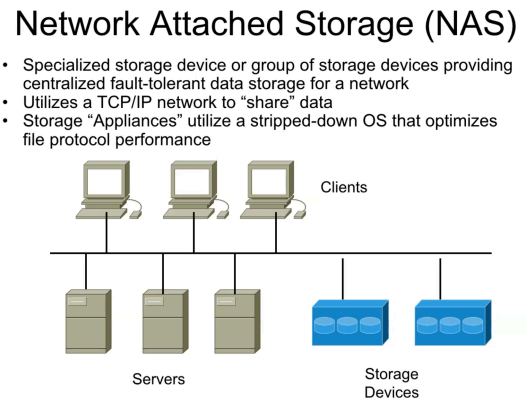


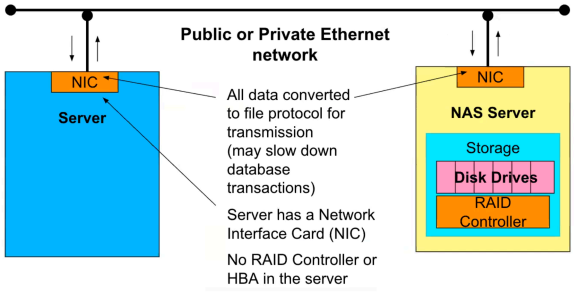
NAS: Network Attached Storage

- 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





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:

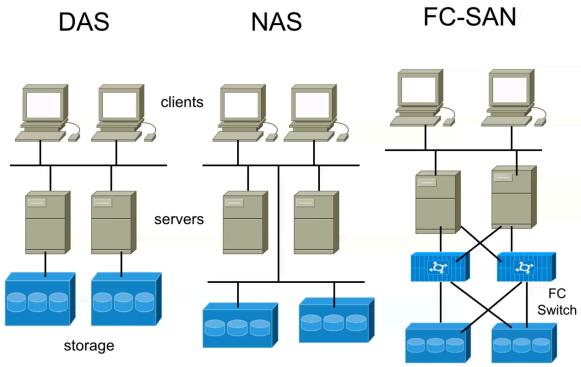
- **Scalability:** Good
- **Availability:** Good - Predicated on the LAN and NAS device working
- **Performance:** Poor - Limited by speed of LAN, traffic conflicts, inefficient protocols
- **Management:** Decent - Centralised management of storage resources
- **Connection:** Homogeneous vs Heterogeneous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

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

General about Object Storage

- More recent
- Not Hierarchical inherently
  - Can mimic hierarchy with how you name objects
  - Backups can take advantage of the mimicked hierarchy
- Quite a saturated market already
  - Amazon S3
  - Microsoft Azure
  - HP Cloud Object Storage
  - etc.
- Open Source Solutions
  - OpenStack Swift (Python)
  - Ceph
  - Riak CS
  - etc.

Before

- Relational databases (SQL)
- NoSQL databases
- Block Storage
  - SAN
  - Oldest and simplest
  - Fixed-sized chunks
  - Block is a portion of data
  - Address: identify part of block
    - No block metadata
    - Limits scalability
  - Good performance with local app and storage
    - More latency the further apart
- File Storage
  - NAS
  - Hierarchical file system
  - Works well with *smaller files*
  - Issues when retrieving large amounts of data
    - Not scalable
  - Unique address → finite number of files can be stored

> ⚠ **Possible Report Weakness**
>
> Make sure to understand the difference between block and file storage

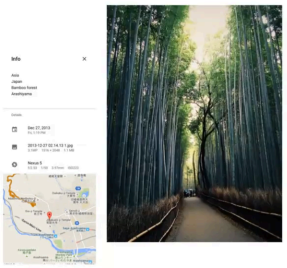
What is an Object

- File + Metadata

Example

File:
15x6x3_10x00x41741x64561
_21x58x511_n.jpg

Metadata:
Image size: 1516x2048
Date taken: 2013-12-27 13:19
Tags: Asia,
Japan,
Bamboo forest,
Arashiyama
GPS: 35.016520, 135.670436



What Object Storage

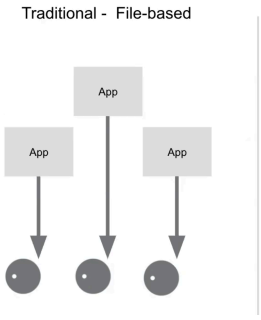
- Collection of *a lot* of objects
- Enables metadata search (kinda similar capability to a database)

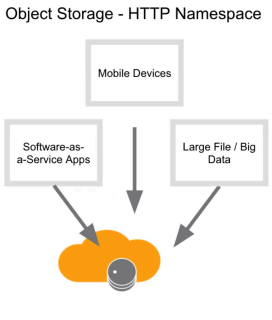
Why Object Storage

Enables loading large amounts of data

Benefits:

- Big data
- Capacity
- Scalability
- Cheap





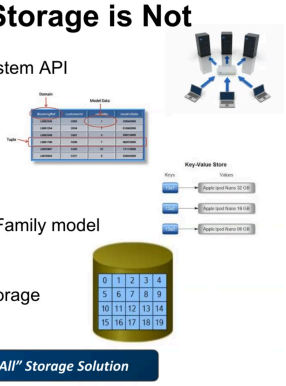
	Object	File	Block
Price	$	$$	$$$
Data-type	Semi + Unstructured	Structured, Semi, Unstructured	Structured
Scales	Billions of Objects and Multi-Exabytes	Millions of Files and 1 Petabyte (ish)	Thousands of files and many TB
Protocols	HTTP REST APIs (e.g. S3, OpenStack Swift)	File Protocols (e.g. CIFS/NFS)	Block protocols (e.g. iSCSI, FC)
Optimized for	• Capacity • Scalability • Eventual consistency • Web accessibility • Broad geo distribution	• Fast file-sharing • Fast file-serving • File-locking • Canonical true files	• Fast random lookups • Reads and writes on small records • Storage for hypervisors
Typical apps	Everything under "File" + gene sequences, video, log files, photos, web content, large data sets	Office automation, design automation, collaborative engineering, word processing docs, presentation graphics	Transactional apps such as ERP, CRM, databases
Approach	An object = a File + all associated metadata + a globally unique identifier	File is stored in directory structure. Limited metadata stored in the file system itself (separate from file)	File is written in "blocks" on spinning media

Web-scale
Site-scale
Application-scale

What is Object Storage NOT

What Object Storage is Not

- Distributed File System
  - Does not provide POSIX file system API support
- Relational Database
  - Does not support ACID semantics
- NoSQL Data Store
  - Not built on the Key-Value/Document/Column-Family model
- Block Storage System
  - Does not provide block-level storage service



Not a "One Size Fits All" Storage Solution

OpenStack Swift

- Can do some erasure coding → think about it as systematic Reed Solomon
- No master/slave coordinator/worker architecture
- One extra level: Container (bucket)
  - Container is a collection of objects
  - Container can have metadata
- Replica management similar to Hadoop
  - With a crawler/scanner that checks for consistency
  - Takes action if inconsistency is found
- Scales linearly

Swift API accessible via HTTP

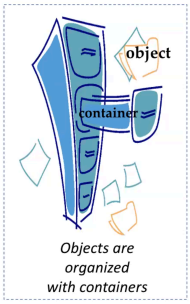
- Applications can consume storage from anywhere
- Larger range of functionality
- Puts the developer in control

Swift Object Storage superior to Traditional Storage

- Designed to scale from TB -> PB -> EB
- Replication is automatic across Nodes, Zones, and Regions
- Supports both Replicas & Erasure Coding
- No Single Points of Failure -> Lose Nodes, Racks or Data Centers
- Ingress / Egress data from all Proxies - No Masters
- Balance Heterogeneous Commodity Hardware

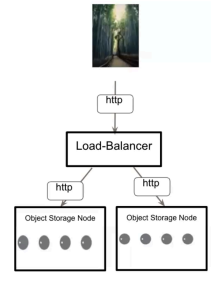
OpenStack Swift Overview

- Uses container model for grouping objects with like characteristics
  - Objects are identified by their paths and have user-defined metadata associated with them
- Accessed via RESTful interface
  - GET, PUT, DELETE
- Built upon standard hardware and highly scalable
  - Cost effective, efficient
- Eventually consistent
  - Designed for availability, partition tolerance



Objects are organized with containers

Access Method: RESTful HTTP API



There are also API operations for Accounts and Containers

Object Storage API Operations for Objects:

Method	Description
GET	Downloads an object with its metadata
PUT	Creates new object with specified data content and metadata
COPY	Copies an object to another object
DELETE	Deletes an object
HEAD	Shows object metadata
POST	Creates or updates object metadata

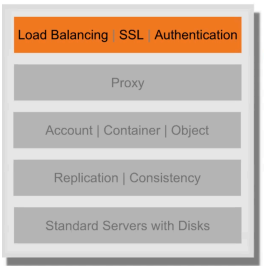
Every Object Has a URL

- Each account has its own URL
- Swift is multi-tenant

- Namespaces used to group objects within an account
- **Containers** are unlimited
- **Like folders, but can't nest them**

- Each object is addressed as a URL
- Users name the object
- **Objects are not organized based on hierarchy**
- Instead, object names may contain "/", so pseudo-nested directories are possible

Swift Architecture

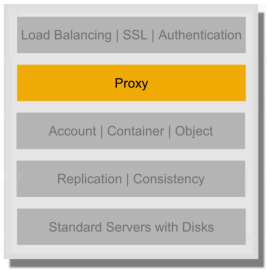


Load Balancing
- Requests are load balanced across all nodes running proxy server processes

SSL
- Optionally, SSL termination can be enabled to encrypt data in flight

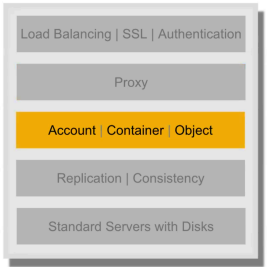
Authentication
- OpenStack Swift has a pluggable authentication system
- Modules include API/UI-driven, LDAP, AD, Keystone

Swift Architecture



Proxy
- Only part of the cluster that "talks" to external clients
- Primarily with HTTP RESTful Swift API
- Routes requests from clients to disk
- Three replicas are simultaneously written
- Quorum required
- Uses single replica for reads
- Routes around failures
- Enforces ACLs set by user

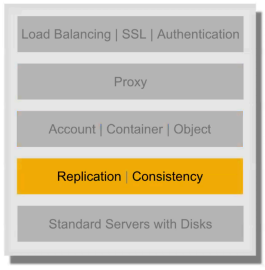
Swift Architecture



Account / Container
- Accounts keep records of containers
- Containers keep records of objects

Object
- The object servers store the data on disk
- Metadata is stored with the data
- Uses standard filesystem (XFS)

Swift Architecture

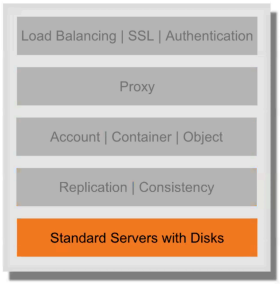


Replication
- Constantly checking for replicas status
- Only updates other replica sites, does not pull in newer versions of objects

Consistency
- Constantly 'scrubbing' data to check for bad data

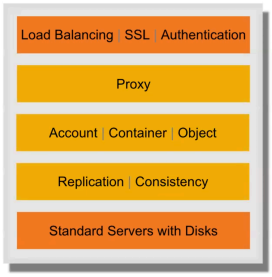
Note: These processes run in the background on Nodes where account, container, or object server processes are running.

Swift Architecture



Standard Servers
- Runs on standard server hardware
- SATA or SAS disks
- No RAID
  - Visibility to hardware beneath
  - Swift is already providing data redundancy
  - Reduce cost

Swift Architecture Summary



- Native HTTP API
- Scales Linearly
- No Single-point of Failure
- Standard Servers and Linux
- Extremely Durable
- Resilient to hardware failures
- Routes around network failures
- Consistency Model Enables Multi-Data Center

Regions and Zones

- Enables handling of very different data-center topologies

Regions:

- Separate physical locations
- Minimum of 1 regions (1 data center)
- A region can be defined within data centers, but is typically not used

Zones:

- Independent failure domains
- Minimum of 1 zone per region
- Zones are typically divisions within a data center
- Failure domains
  - Maybe physical fire walls
  - Maybe different power sources
  - Maybe different network switches
  - Maybe different racks

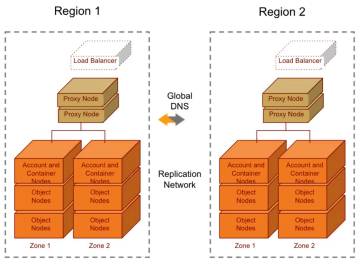
Cluster:

- A collection of regions
- Make these global for optimal reliability
- Specific regions
  - Can have different policies
  - Like different replication policies
  - Different erasure coding policies

Differences between Hadoop and Swift:

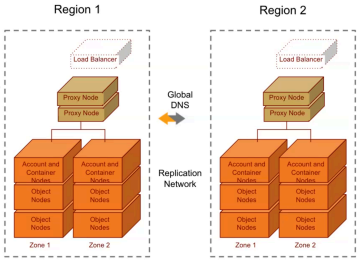
- Hadoop verified to the client right after receiving the data
  - Fast!
- Swift only tells client stuff is done after complete storage is finished
  - Slow!
  - But more reliable
- Hadoop only spread on two racks
  - Two racks fail → data is lost
  - Faster! (Than Swift)
- Swift object clusters
  - 1 replica on 3 different racks in different zones in different regions
  - Slower! (Than Hadoop)
  - Highly reliable
  - Spread out as much as possible

Regions
- Physically separate, often defined by geographical boundaries
- Minimum: 1 Region



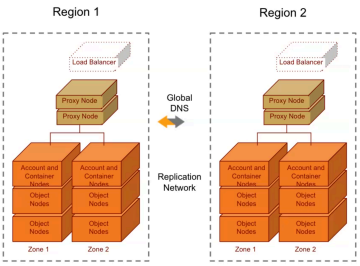
Regions and Zones

Zones
- Regions contain one or more zones
- Designate a group of nodes sharing a set of physical hardware
- Also referred to as failure domains



Regions and Zones

Multi-Region Clusters
- Goal: Tolerate failures across regions and zones
- Policies control placement across regions
- Send requests to "closest" region

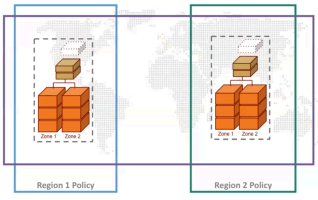


Storage Policies

Benefits
- Optimizes storage for applications and users
- Consolidates storage tiers under one system
- Simplifies management
- Lowers storage infrastructure TCO

Policies Encompass:
- Storage media
- Number of replicas
- Erasure Codes / Replicas

Common Policy Groupings:
- According to performance
- According to geography or political boundary
- According to data protection scheme, e.g., number of replicas or erasure coding



When to use what

- Hybrid options are best!

Use Case – Big Data / MapReduce

Hadoop / Spark
Philosophy: Let HDFS do what it's best at:
Serving data where you want it when you need it
Swift for warm and cold data storage. Why?

- Durability and reliability guarantees
- Managed capacity: Grow to and beyond petabytes
- Easier integration with various data input sources
- Share results using a common storage platform

Example: Run MapReduce job

- Read input data from SwiftStack
- Write transient results to HDFS
- Write result to Swift

⚠ Possible Report Weakness

Replication placement strategies still apply to object storage
→ Doesn't depend on the type of storage
→ Simply just strategies for how to place replicas

11: Compression & Delta Encoding

Basic Compression

- Reduction of the size of the data
- Alias: Source Encoding

Idea

- Represent frequent repetitions of information with a shorter representation
- Can think of some information as occurring with some probability

Information of an event given that it has a probability of p is

$$I = \log_2\left(\frac{1}{p}\right) = -\log_2(p) \tag{4}$$

- **Provides nice properties:**
  - Information of independent events is additive, but probabilities are multiplicative

$$I_A + I_B = -\log_2(p_A) - \log_2(p_B) = -\log_2(p_A p_B) \tag{5}$$

Shannon Entropy

- Average information of a random variable

$$E[I] = H(p_1, p_2, ..., p_n) = \sum p_i \log_2\left(\frac{1}{p_i}\right) \tag{6}$$

- **Another way to about it:**
  → Expected uncertainty associated with this set of events
  - H is non-negative
  - $H \leq \log_2(N)$, where N is the number of events (equality if all events are equally likely)
- Most compression with low entropy (high certainty, highly predictable)

Types of Compression

Lossless:

- No information is lost
- Can be reversed exactly
- Typically used for text, programs, data
- Achieves lower compression ratios

Lossy:

- Some information is lost
- Cannot be reversed exactly
- Typically used for images, audio, video
- Achieves higher compression ratios

Entropy

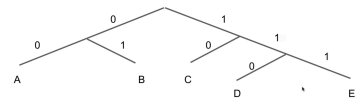
- Expected amount of information in a message
- Lower bound on the amount of information that must be sent
- **Computable**
  - If you know the probability of each symbol

Compression Schemes

Shannon-Fano & Huffman Coding

Encoding for the Shannon-Fano Algorithm:

1. Sort symbols according to their frequencies/probabilities: ABCDE
2. Recursively divide into two parts, each with approx. same number of counts



Bits in average: 2 x (15 + 7 + 6)/(39) + 3 x (6 + 5)/39 ~ 2,28 bits/symbols

This is a basic information theoretic algorithm

Symbol	A	B	C	D	E
Count	15	7	6	6	5

Huffman Coding

Based on the probability (or frequency of occurrence) of a data item

The principle is to use a lower number of bits to encode the data that occurs more frequently

Codes are stored in a **Code Book**

The code book + encoded data must be transmitted to enable decoding

Fixed # of bits to variable # of bits

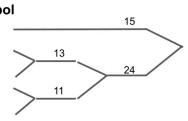
Huffman Coding

1. **Init:** Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE)
2. Repeat until the OPEN list has only one node left:
   a. From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them
   b. Assign the sum of the children's frequencies/ probabilities to the parent node and insert it into OPEN
   c. Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN

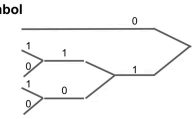
Huffman Coding

- **Unique Prefix Property**:
  - No code is a prefix to any other code
  - Great for decoder, unambiguous
- If prior statistics are available and accurate, then Huffman coding is very good

Count	**Symbol**
15	A
7	B
6	C
6	D
5	E



Count	**Symbol**	**Code**
15	A	0
7	B	111
6	C	110
6	D	101
5	E	100



Bits in average: 1 x 15/(39) + 3 x (24)/39 ~ 2,2307 bits/symbol

Entropy:
- 15/39 x $\log_2$(15/39) - 7/39 x $\log_2$(7/39) - 2x6/39 x $\log_2$(6/39) - 5/39 x $\log_2$(5/39)
= 0,5301967 + 0,44477772 + 0,8309045 + 0,3799325
= **2,18581142**

LZW: Lempel-Ziv-Welch

- ZIP based on LZW
- Automatically discovers inter-symbol relations
- Builds the code-book on the fly

Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique

Example: Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries

Why not just transmit each word as an 18 bit number?

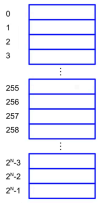
- **Solution**: Find a way to build the dictionary adaptively
  - Original methods due to Ziv and Lempel in 1977 and 1978. Terry Welch improved the scheme in 1984 (called LZW compression).
  - Variable length to fixed length ("reverse" of Huffmann)

LZW Compression Algorithm

```
w = get input symbol;
while ( there are still input symbols )
  {
   k = read a character
   if w + k exists in the dictionary
      w = w + k;
   else
      add w+k to the dictionary;
      output the code for w;
      w = k;
  }
Output the code for w          (+ = concatenate)
```

LZW Compression

- Table size: 2^N
- N bits to represent each index
- Send **table indexes**
- String table can be reconstructed by the decoder using information from the encoded stream → table is not sent



abbbabbbab

10 symbols of 8 bits each → 80 bit sequence

```
w = get input symbol;
while ( there are still input symbols )
  {
   k = read a character
   if w + k exists in the dictionary
      w = w + k;
   else
      add w+k to the dictionary;
      output the code for w;
      w = k;
  }
```

As long as 6N < 80 → compression

N < 40/3 ~13,333 bits

Ex: N = 12 bits → Rate = 6 x 12 / 80 = 0,9

N_{min} = 9 bits → Rate = 6 x 9 / 80 = 0,675

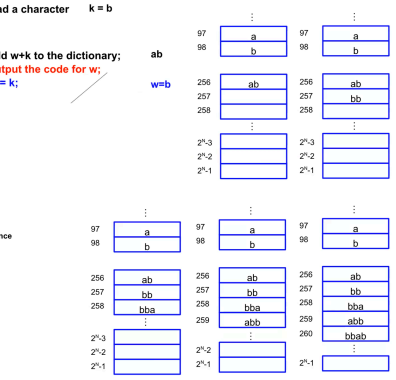
Output: 97, 98, 257, 256, 258, 98

6 symbols of N bits each → 6N bit sequence

abbbabbbab

w = get input symbol; **w = a**
k = read a character **k = b**

else
add w+k to the dictionary; **ab**
output the code for w;
w = k; **w=b**
}



Output: 97, 98, 257, 256, 258, 98

From 10 to 6 symbols

JPEG

- Lossy

Repition suppresion

- Counting the number of times a symbol is repeated when repeated

Application:

- Silence in audio
- Spaces in text
- Background in images

Run-length Encoding

- Specific version of repetition suppression

- Used in JPEG in some cases
- Encoding would be heavier than original if no repetition

Example:
- Original: `AAAAAABBBBCCCCC`
- Encoding: `(A, 6), (B, 4), (C, 5)`

Delta Encoding

- If you know you get different versions of that the files
- Sequences as compared to previous versions

Applications:
- Version control (`git`)

The Problem
- Storing each version independently creates massive unnecessary redundancy
- No inherent way of keeping track of the changes/dependencies

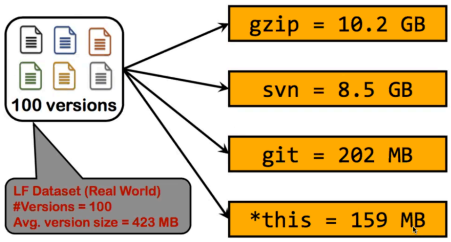
Solution
1. Store the first version
2. Store the differences between the first and second version
3. Repeat for each version

How to access any version:
1. Start with the first version
2. Apply the differences in order

Example
Git/SVN is optimised for text files
- Uses large amounts of RAM for bit files

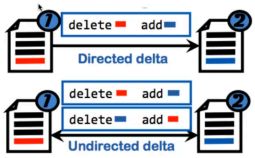
Slides

Version Control Systems in Practice

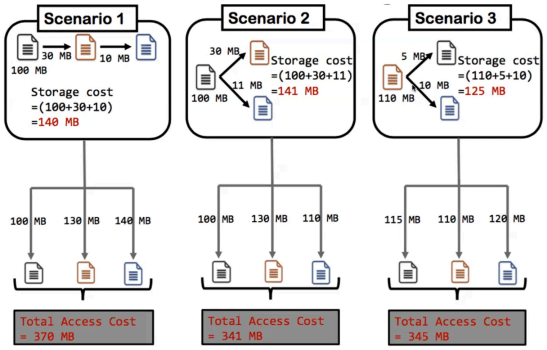


How to recreate a version?

Use **deltas:** A delta between versions is a file which allows constructing one version given the other



Storage/Recreation Tradeoff (with delta encoding)



Problem

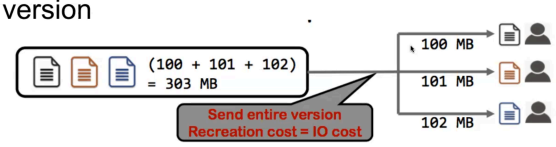
Find a storage solution that:

- Minimizes (total) recreation cost within storage budget
- Minimizes maximum recreation cost within a storage budget

Costs

Storage Cost: Space to store all versions

Recreation Cost: time required to access a version



VCDIFF (RFC 3284)

COPY(t,p) defines the *t* number of bytes to use starting from position *p* in the dictionary/source

ADD(t,s) indicates the *t* number of bytes to add to the dictionary (and the new file) with a sequence *s* (s has t bytes)

RUN (x,s): A byte b will be repeated x times

File V_1: 0,1, 2, 3, 4, 5, 6, 7, 8, 9
can be a **source** in VCDIFF terms (a prior dictionary)

File V_2: 0, 1, 2, 7, 8, 9, 3, 4, 5, 6, F, D
arrives, the delta representation of this file is
{COPY (3,0), COPY(3,7), COPY(4,3), ADD(2, {F,D}) }

→ Recognized two sequences of 3 bytes & one of 4 bytes that could be used in the delta representation

Source:
a b c d e f g h i j k l m n o p
Target:
a b c d w x y z e f g h e f g h e f g h **z z z z**

COPY 4, 0
ADD 4, w x y z
COPY 4, 4
COPY 12, 24
RUN 4, z

12: Data Deduplication

Data Deduplication

- If you already have data stored that

Cost of Data

- File size itself
- Cost of redundancy

Two Questions

- Can we reduce storage costs
- Can we reduce the cost of redundancy

Example

- Send and email to 100 employees with a 10MB attachment
- What is the cost of sending this email?
- $10\text{MB} \times 100 + 100 \times \text{rest of mail} = 1\text{GB} + 100 \times \text{rest of mail}$ (without redundancy)
- $10\text{MB} \times 3 \times 100 + 100 \times \text{rest of mail} = 3\text{GB} + 100 \times \text{rest of mail}$ (with redundancy)

Instead:

- Store the 10MB attachment once
- $10\text{MB} + \text{overhead} + 100 \times \text{rest of mail}$

Idea

- Instead of doing deduplication at file level
- Recognise bit-patterns within files and deduplicate at bit-level
- **Or at a *block level* ← This is the one used (Fixed size block, variable-sized is very uncommon and difficult)**
- Do this across all files
  - Think of it as lego, going from the finished build to the list of bricks
  - Store the list of bricks and *the instructions*
- The more data you store, the more opportunities for deduplication
- Instruction are typically an index to a table containing a pointer to the data

Goal:

- Eliminate storage of data with same content

Challenges

- Can we preserve performance?
- Can we support general file system operations?
  - Read, write, modify, delete
- Can we deploy deduplication on low-cost commodity hardware?
  - A few GB of RAM, 32/64-bit CPU, standard OS
  - Maybe a small server
  - Or you want each node to be part of computation

Practical Ideas

- **Deduplication in backup systems**
  - Assume no modifications
- **Deduplication file systems**

- ZFA, OpenDedup SDFS
- Consumes significant memory

- **VM image storage?**
  - VM images are large and have a lot of common data

Compare Blocks

- Don't compare every single block
- Cryptographic hashing
  - MD5, SHA-1
  - Hashing is fast
  - Probability of getting different content with same hash is *very low*
    - Combat by having larger hashes

How often do I get matches?

- Birthday paradox
- N people, what is the probability that two people have the same birthday?
- Number of values to test to have a probability of at least one collision greater than 50% is $N^{1/2}$ for N possible values
  $\rightarrow$ Block size of $N = 2^n$
  $\rightarrow$ Need $2^{n/2}$ values to have a 50% chance of a collision
- Above is *worst case* with equal probabilities for each block
  $\rightarrow$ typically there is *more correlation* in data

When to Delete Blocks

- When no file references it
  - This requires exploring/analysing all files to find references

Instead:

- Keep count of how many times a block is referenced
- This should be part of the initial processing
- Decrement count for all blocks referenced by a file when it is deleted
- This count gives a notion of *importance* of a block

Where to put index structure

RAM

- Hadoop, ZFS
- Fast, but limited by RAM size
- Per 1TB of data, around 4GB of RAM is needed

Disk

- Updating each data block moves the disk head
  - Impacts performance $\rightarrow$ Slow

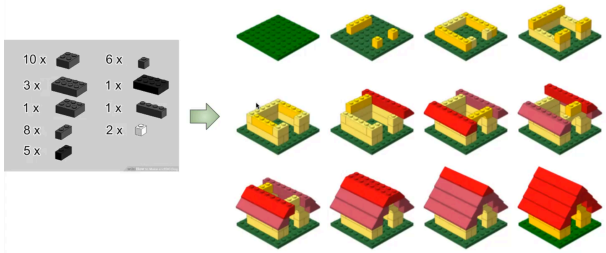
LiveDFS

- Partial storage of fingerprints in memory
- Rest on disk
- Balance between RAM and disk
- Per 1TB of data, around 1.6GB of RAM is needed

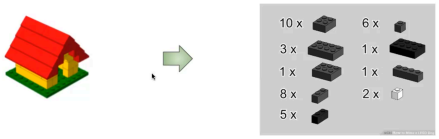
Important Slides

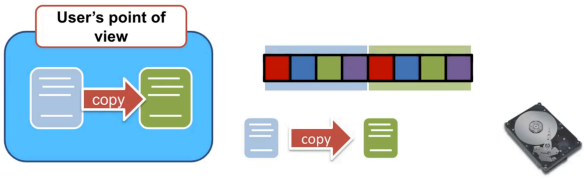
What is Deduplication?

Think of it as a LEGO problem

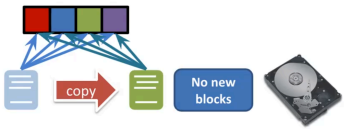


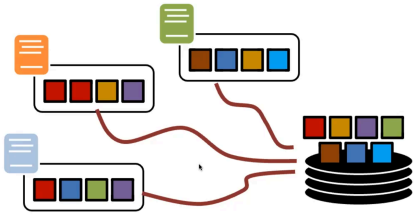
But you can also take the data objects and convert them into pieces:



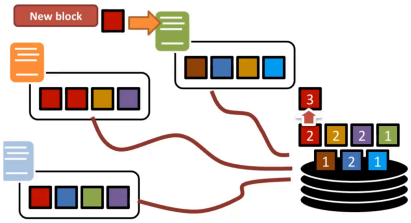


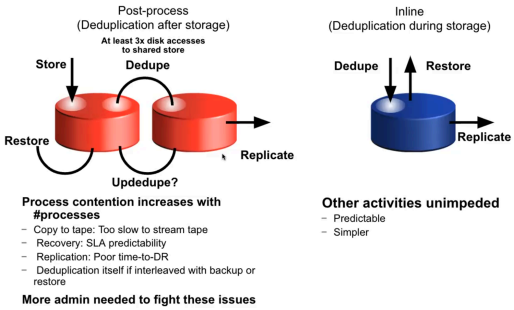
Goal: Eliminate storage of data with same content





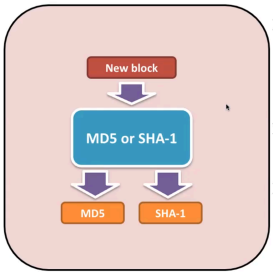
Example: storage space reduced by 5/12 = 42%





Basics: Fingerprints

- How to compare blocks?
- Solution: Use cryptographic hashes (or **fingerprints**)



- Hash-based comparisons
  - Same content → same hash
  - Different content → different hashes with high probability
- Pros: block comparison reduced to hash comparison
- Cons: collision may occur, but with negligible probability

[Quinlan & Dorward, '02]

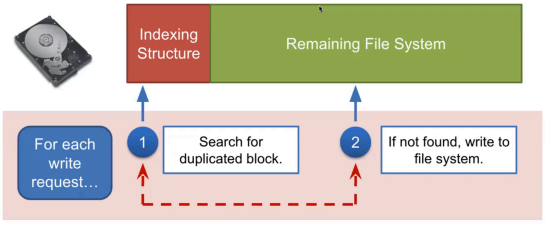
Option 1: Index Structure in RAM

- How about putting whole index structure **in RAM**?
  - Used in existing dedup file systems (e.g., ZFS, OpenDedup)
- Challenge: need large amount of RAM
- Example: per 1TB of disk content

Block Size	4KB
Using MD5 checksum	16 bytes per block
Size of Index	1TB / 4KB x 16 bytes = **4GB**

Option 2: Index Structure on Disk

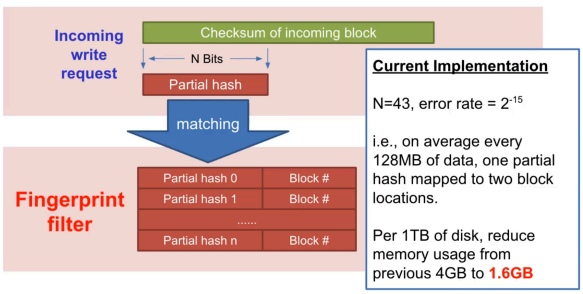
- How about putting whole index structure **on disk**?



- Challenge: updating each data block and its index keeps the disk head moving, which hurts performance

Example: LiveDFS Design

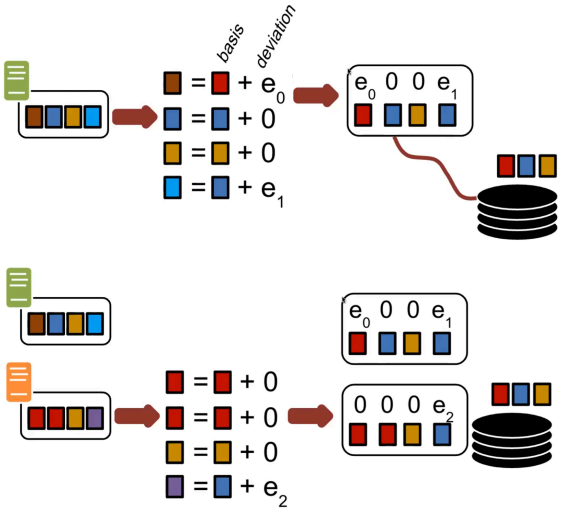
- Store partial fingerprints in memory
  - Infer if same block exists, and where it is "potentially" located



Generalised View

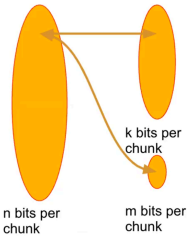
- Block basis + deviation
- Maximise similarities in data
- Gains start earlier
- Easier to find matches

Important Slides



HOW TO DO THE TRANSFORMATION?

- Promotes large amount of matches: n +1 matches
- Small deviation (few bits): m bits
- There are a number of options with better characteristics



If m = 15 bits

n ~ 4 KB

32768 matches

Example of Transformation

Error Correcting Codes: case where code can correct errors

Data 1 0 1 0 → Add redundancy 1 0 1 0 1 1 0 → Channel with errors 1 0 1 1 1 1 0 → Recovered data 1 0 1 0

In our case...

Data	Map (basis + deviation)
1 0 1 0 0 1 0	S = bit 4
1 0 1 1 1 1 0	S = bit 3
1 0 1 0 1 1 0	1 0 1 0 + S = no error
1 0 1 0 1 1 0	S = bit 2
0 0 1 0 1 1 0	S = bit 0

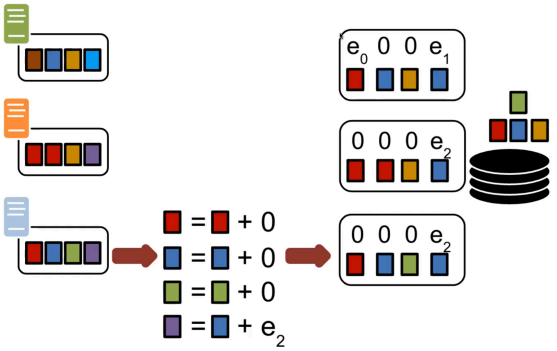
m bit for deviation

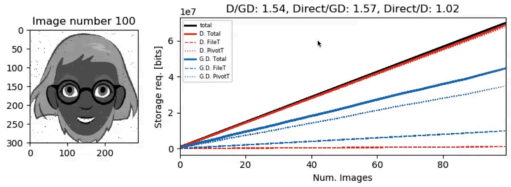
$n = 2^m - 1$ bits

$k = 2^m - m - 1$ bits

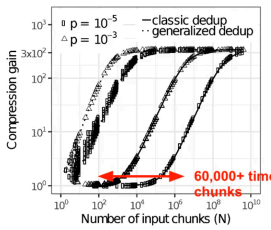
n + 1 sequences matched to 1 basis

Compression gain is 12/4 = 3

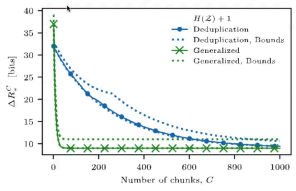




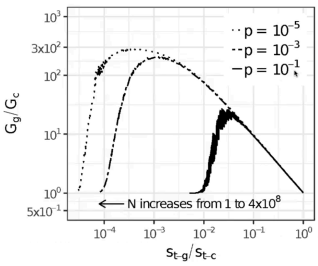
Compression Starts Earlier



Information Theory – proven this speed up



Need Smaller Registry



Size of the registry gen dedup vs dedup

ZFS: rule of thumb 5GB of Memory for every 1TB of data

Can lower this requirement (e.g. 10 – 10000 times)

13: Fog/Edge Storage

Fog/Edge Storage

- Mobile devices
- Storage and delivery of content in peer devices at the network edge
  - Potential reduction in latency
  - Potential increase of data rates

Mobile

- Each device generates some data
- If a device is lost, the data from that device is lost if nothing clever is done
  - Other neighbouring devices can store the data
- Replicas bad, because of storage and bandwidth costs
  - Repairing a lost device is *very* expensive
- Erasure codes are better
  - Model as multicast flow problem, see slides

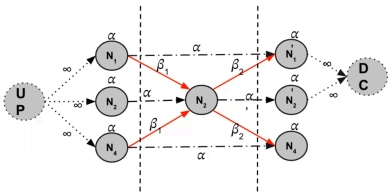
MSR Repair: Repair with no new-comer nodes

- See slides

Fog/Mobile Storage

(a) Data collection at individual devices, sharing & distributed storage using other IoT devices

It can be modeled as a multicast flow problem

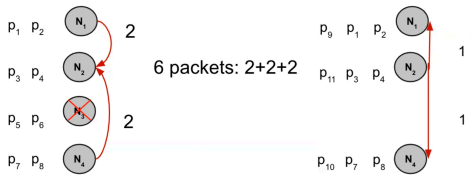


MSR Repair: No newcomer nodes

$A_1\ A_2\ A_3\ A_4\ A_5\ A_6$

8 Coded packets are produced

$p_1\ p_2\ p_3\ \cdots\ p_7\ p_8$



When using RLNC: if the finite field is large, all p_i are linear independent with high probability

Redundancy	Total Transferred
33%	100%

Before repair

$$(\alpha, \gamma) = \left(\frac{B}{k}, B\right)$$

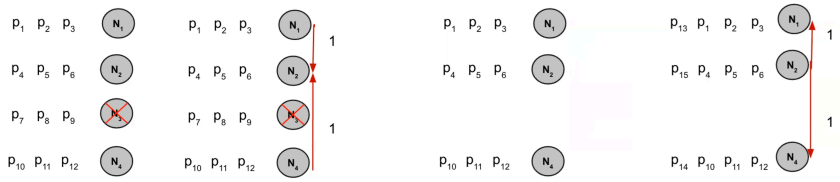
After repair

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

$A_1\ A_2\ A_3\ A_4\ A_5\ A_6$

12 Coded packets are produced

$p_1\ p_2\ p_3\ \cdots\ p_{11}\ p_{12}$



5 dimensions: 3+1+1

Recodes, forming: p_{13}, p_{14}, p_{15}

$$p' = \sum_{i=1}^{l} c_i \cdot p_i$$

4 transmissions total between nodes

When using RLNC: if the finite field is large, all p_i are linear independent with high probability

Redundancy	Total Transferred
100%	66%

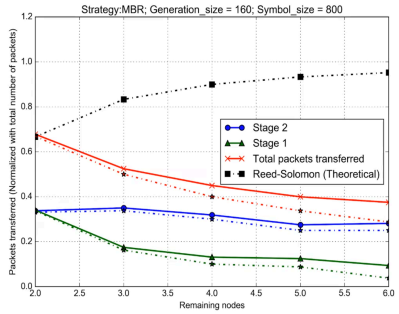
Before repair

$$(\alpha, \gamma) = \left(\frac{2B}{k+1}, \frac{2B}{k+1}\right)$$

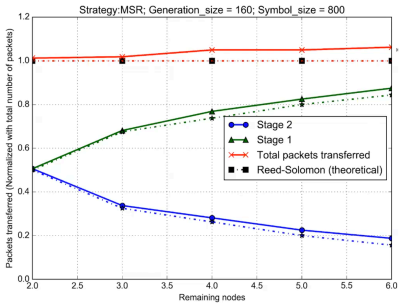
After repair

$$(\alpha', \gamma') = \left(\frac{2B}{k}, \frac{2B}{k}\right)$$

Fog/Mobile Storage: MBR



Fog/Mobile Storage: MSR



⚠ Possible Report Weakness

What does “min-cut” mean for multicast flow problems?

14: Security in Storage Systems