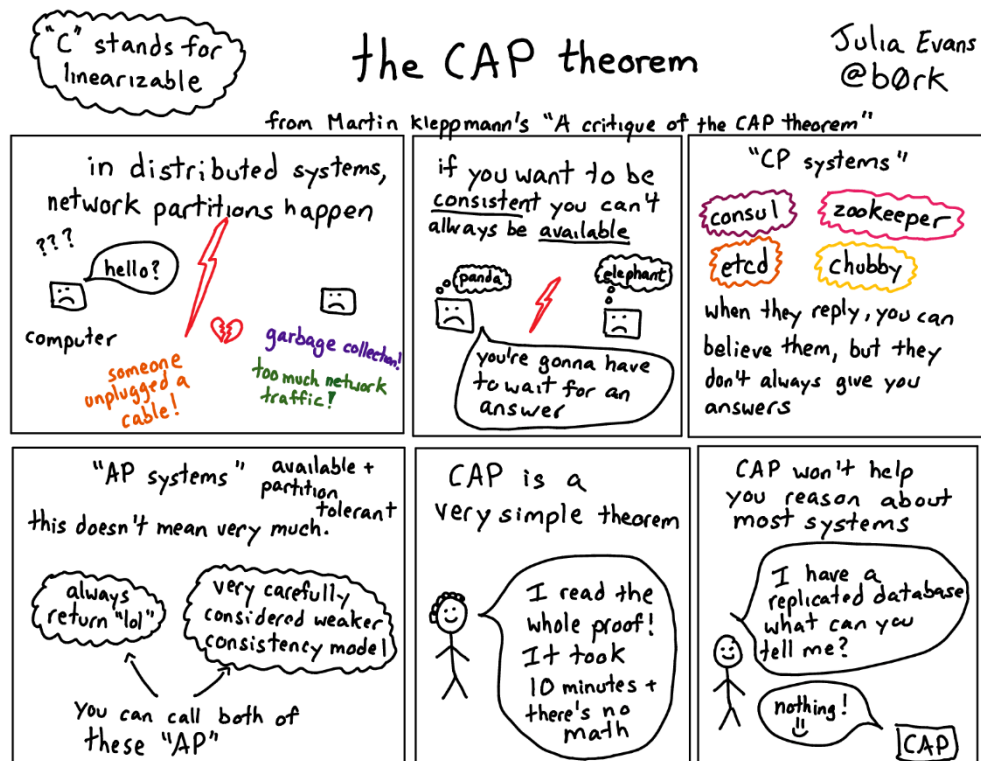


CAP Theorem

Consistency – dictionary.com

- steadfast adherence to the same principles, course, form, etc.:
- agreement, harmony, or compatibility, especially correspondence or uniformity among the parts of a complex thing;
- the condition of cohering or holding together and retaining form; solidity or firmness.

By Julia Evans Blog –
<https://jvns.ca/blog/2016/11/19/a-critique-of-the-cap-theorem/>

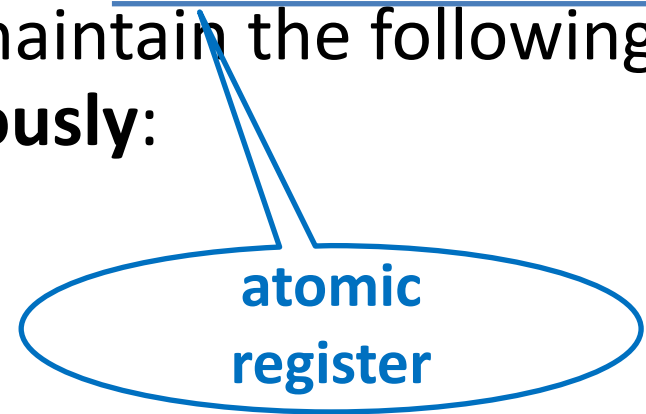


CAP Theorem Introduction

- It is **impossible** for a replicated **read-write store** in an asynchronous system to maintain the following **three guarantees simultaneously**:
 - Consistency
 - Availability
 - Partition-tolerance
- Initially, conjectured by Eric Brewer in 1998, later proven by *Lynch et al.*
- Describes *tradeoffs* involved in distributed system design

CAP Theorem Introduction

- It is **impossible** for a replicated read-write store in an asynchronous system to maintain the following **three guarantees simultaneously**:
 - Consistency
 - Availability
 - Partition-tolerance
- Initially, conjectured by Eric Brewer in 1998, later proven by *Lynch et al.*
- Describes *tradeoffs* involved in distributed system design

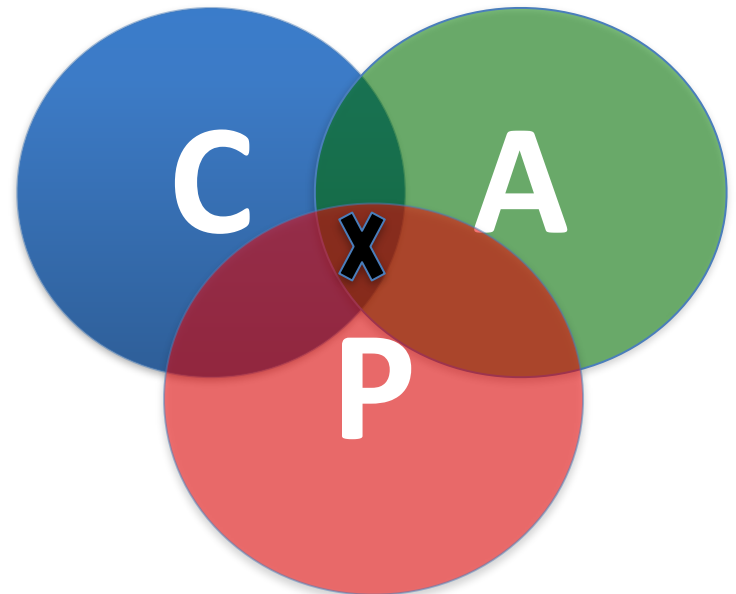


Definition of *Atomic Register*

- Models **replicated read-write store**:
 - `set(X)` sets value of register to X
 - `get()` returns the value of register
- **Replicates and distributes register, must maintain consistency and availability** across all replicas
- Models many distributed system types
 - Key-value store, distributed shared memory
 - Replicated system, distributed file system, etc.

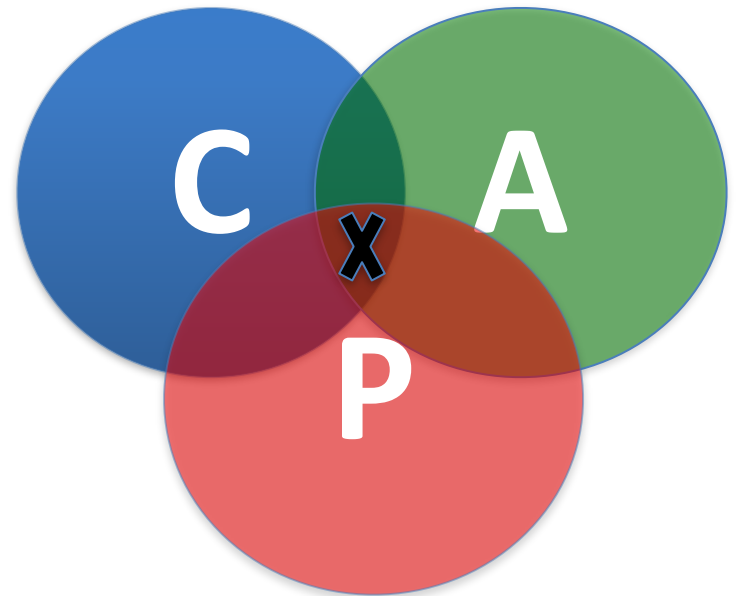
C-A-P: Choose two out of three*

- **Consistency:**
 - All read requests should read the *latest* value (or return an error)
- **Availability:**
 - All requests should *return successfully*
- **Partition-tolerance:**
 - The system can *tolerate* arbitrary number of communication *failures*



C-A-P: Choose two out of three*

- Consistency:
 - All read requests should read the *latest* value (or return an error)
- Availability:
 - All requests should *return successfully*
- Partition-tolerance:
 - The system can *tolerate* arbitrary number of communication *failures*
- Traditional view
 - Today, more a **spectrum**



Definition of Consistency

- Refers to **replication consistency**
 - Not related to $A - C - I D$ properties for transactions
- Could mean **strict consistency**
 - As we know, this is by and large impossible in a distributed system
- Thus, here, assumes **linearizability**
- This usually means replication across sites should be done **eagerly**

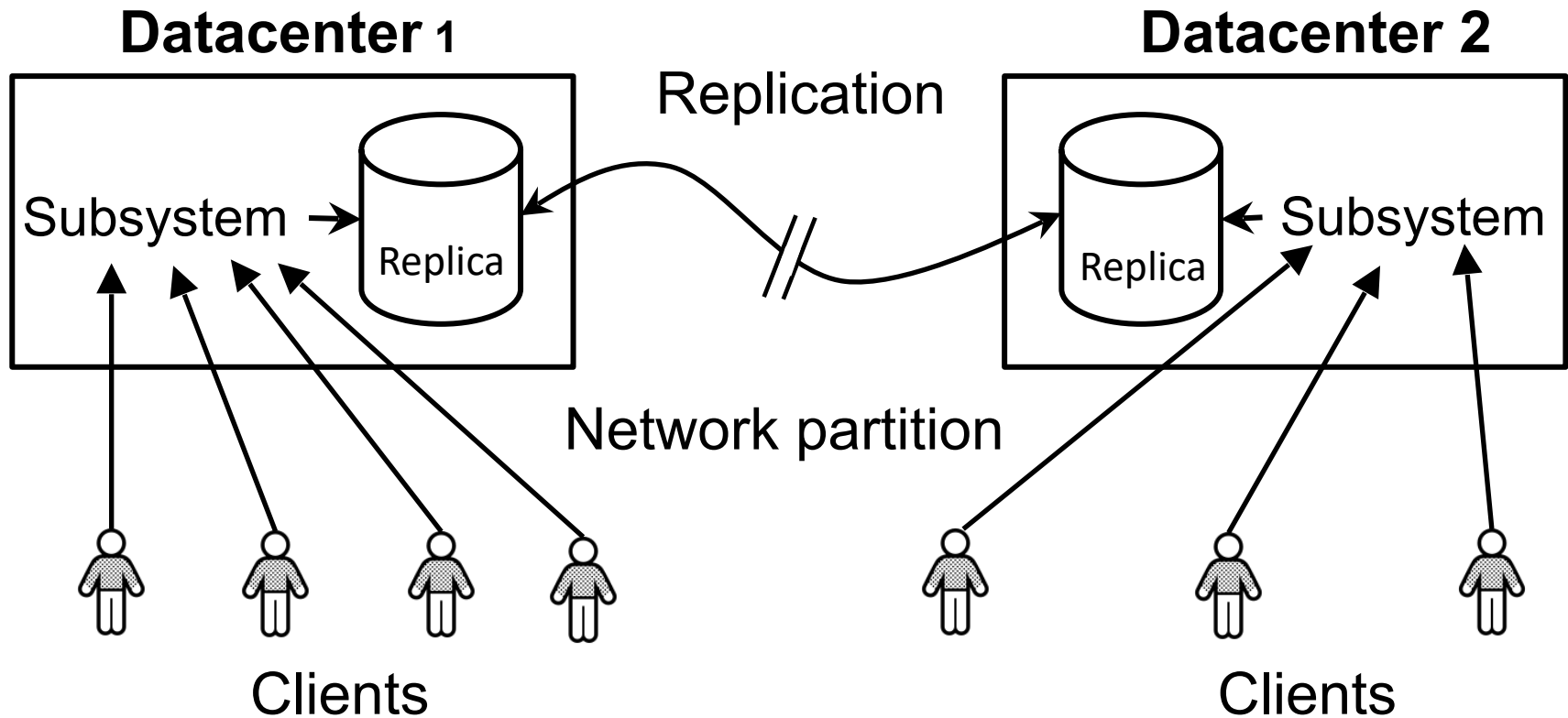
Definition of Availability

- **Every request** received by a non-failed node must result in a **non-error response**
 - **Non-triviality requirement:** a system which always responds with errors is **not available**
- Assumes a **crash failure** model for nodes
 - **Functioning nodes** must continue to operate even if there are **failed nodes** in system
- No requirement on **latency**: response can be very slow but must eventually come through
- Both a **weak and strong** definition: no latency guarantee, but 100% response success

Definition of Partition-Tolerance

- Asynchronous system model
- Message loss (failure model)
- Partition means total communication loss between partitioned subsystems
- Subsystems **continue request processing** even if a network partition causes communication loss within system
- If the system requires a **stronger system model**, or a **weaker failure model**, then it is **not partition-tolerant**
- No guarantee that **partitions recover**, but it doesn't mean they are **always present** either

Definition of Partition-Tolerance



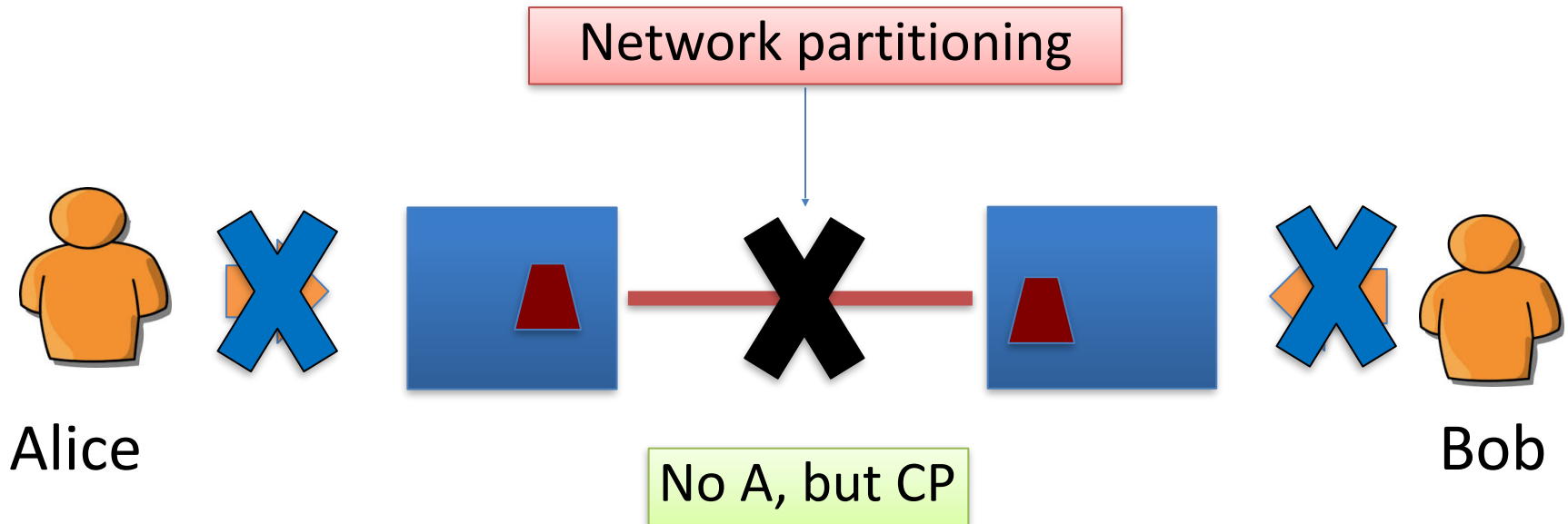
Illustrating Example

Hotel Booking: are we double-booking the same room?



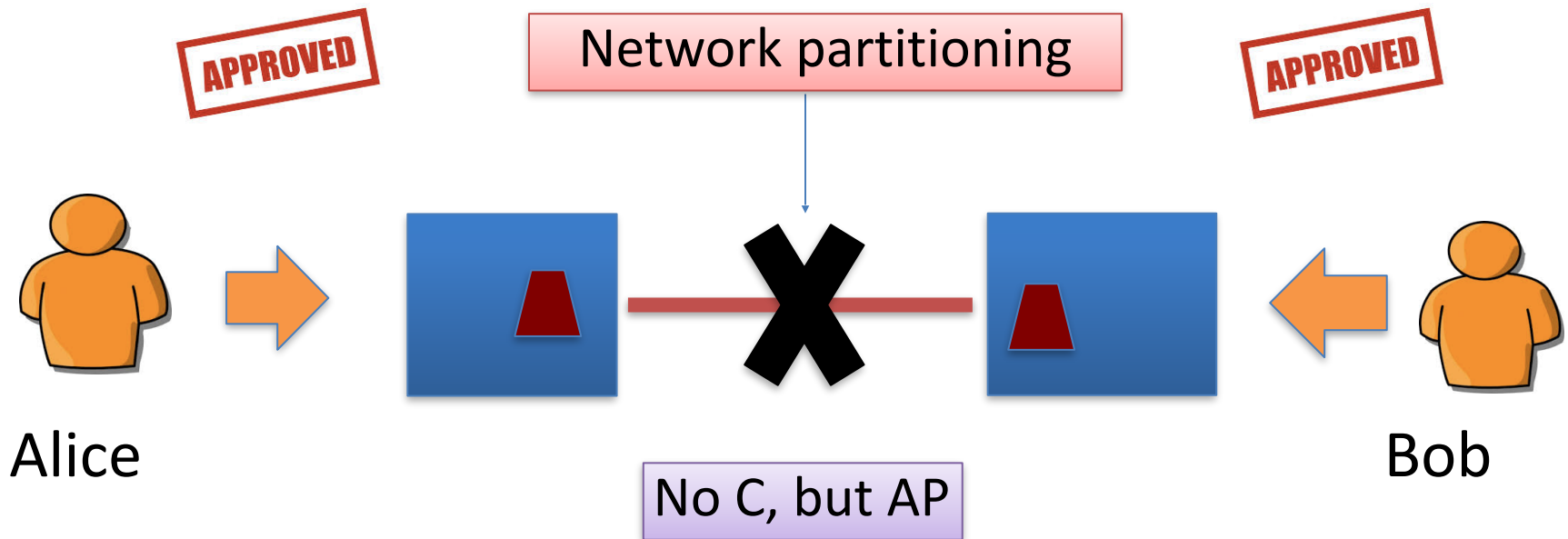
Illustrating Example

Hotel Booking: are we double-booking the same room?



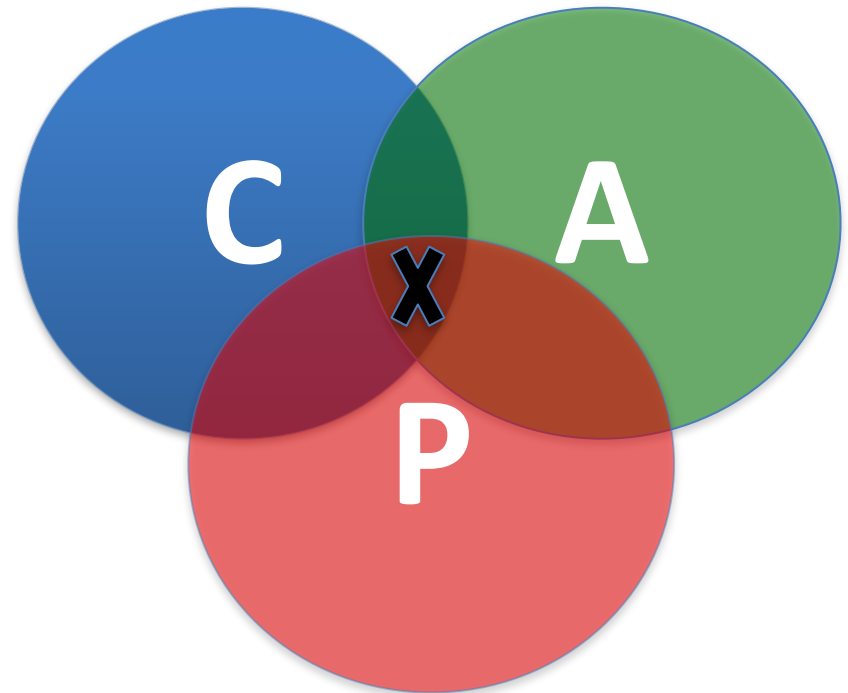
Illustrating Example

Hotel Booking: are we double-booking the same room?

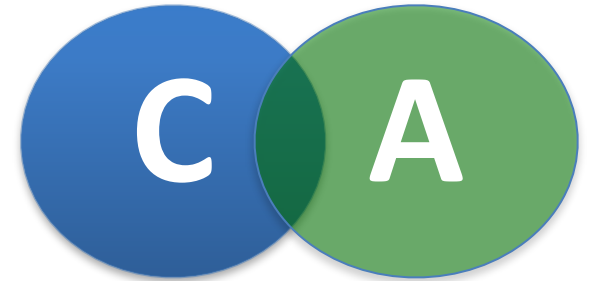


Types of CAP systems

- CAP says “*pick two out of three*”
- The following systems seem possible:
 - CP
 - AP
 - CA
- But it’s not so easy...



CA Systems



- A “perfect” system!
- But with strong assumptions:
 - Either the network is reliable (Fallacy of DS...)
 - Or, the network is not used (then it is not a DS!)
- *“Of the CAP theorem’s Consistency, Availability, and Partition-Tolerance, **Partition Tolerance is mandatory in distributed systems.** You cannot not choose it.”*
 - Coda Hale, Yammer software engineer

Misconception #1: Always choose two

- CA systems cannot be used in practice
- **But**, when there are **no network partitions**, every system can behave like CA!
- In other words, “*choose two*” only takes effect **during network partitions**
- So in reality, there are **only two types of systems** ... i.e., if there is a partition, does the system **give up availability or consistency?**
 - Daniel Abadi, co-founder of Hadapt

Misconception #2: C, A are binary

- CAP theorem uses very narrow definitions of C,A:
 - Consistency: Linearizability
 - Availability: Infinite latency budget, 100% successful
- *“The “2 of 3” formulation was always **misleading** because it tended to oversimplify the tensions among properties. **CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.**”*
 - Eric Brewer, CAP Theorem

Reality of the CAP Theorem

- *“Many designers incorrectly conclude that the theorem imposes certain restrictions on a DDBS **during normal operation** and therefore implement an **unnecessarily limited system**.”*
 - Daniel Abadi, Co-founder of Hadapt
- All systems are, in fact, CAP, but tune how much C and A are provided during P
 - $CP \rightarrow C(a)P$, and $AP \rightarrow (c)AP$
- Provides **freedom** to design system to **suit** application requirements
 - E.g., by choosing appropriate consistency level

AP: Best Effort Consistency

- Example:
 - Web caching (cf., cache consistency)
 - Network File System (cf., concurrent writes)
- Characteristics:
 - Optimistic replication (i.e., lazy replication)
 - Expiration/time-to-live
 - Conflict resolution (e.g., CRDTs)
- Cassandra, Dynamo are AP systems:
 - Eventual consistency: stale data can be read
 - Tunable towards CP

CP: Best Effort Availability

- Example:
 - Distributed lock services (Chubby, ZooKeeper)
 - Paxos (safe, not live)
- Characteristics:
 - Eager replication
 - Pessimistic locking
 - Minority partition becomes unavailable
- BigTable / HBase are CP systems:
 - Provide linearizability
 - Under partitions, cannot access TabletServer/
RegionServer

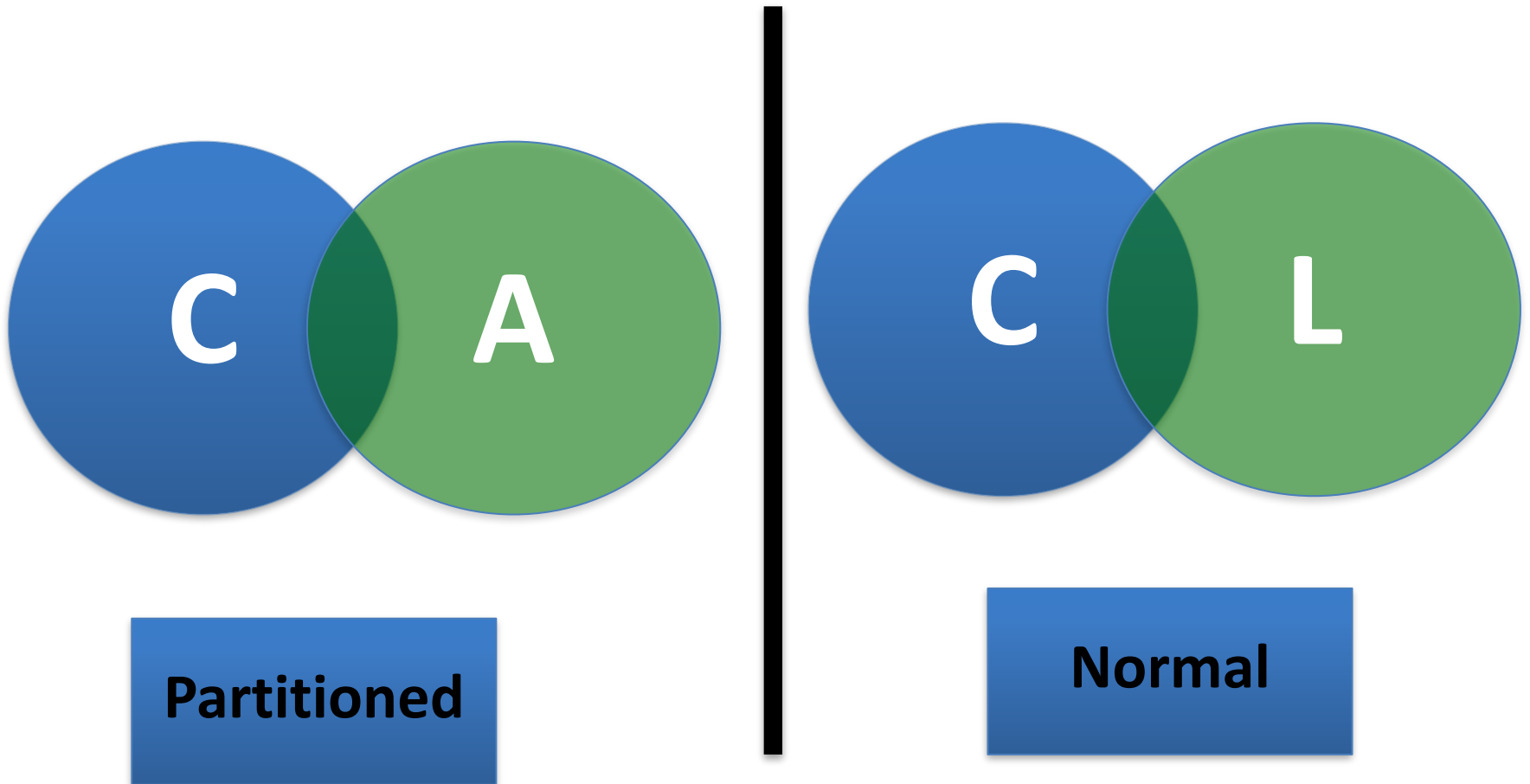
Misconception #3: Static systems

- Systems don't always behave the same way at run-time
- It is possible to design systems which behave differently depending on the operational situation when **partitioning**
- Example: airline reservation system
 - When most seats available: system **behaves AP**, worries about capacity limit later
 - When plane is close to capacity: system **behaves CP**, ensures no overbooking, or **behaves AP** to maximize profit and handle compensations out-of-band

Extended Model: PACELC

- A more complete description of the space of potential tradeoffs for distributed system:
 - If there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?
- Original CAP theorem ignores latency, yet important in practice

PAC/ELC



Examples

- PA/EL: Give up consistency at all times for availability and lower latency
 - Dynamo, Cassandra (tuneable), Riak, web caching
- PC/EC: Refuse to give up consistency, pay the cost in availability and latency
 - BigTable, HBase, VoltDB/H-Store
- PA/EC: Give up consistency when partitions occur, but keep consistency during normal operation
 - MongoDB
- PC/EL: Keep consistency when partitions occur, but give up consistency for latency during normal operations
 - Yahoo! PNUTS

Summary CAP Theorem

- Classically described as “*pick two out of three*”: **consistency**, **availability**, **partition-tolerance**
- Really boils down to **choosing C or A**, since P is a **must-have** for practical systems
- During normal operations, systems can all be **CA**
- Only concerns “**perfect**” **notions** of C and A
- In reality, C and A **are tunable**, systems tend to maintain C and A during P to some degree
- Systems can **adapt** dynamically to become AP or CP for different operational situations
- PACELC: extended model which considers differentiations between P and not P and consistency vs. latency

Self-study Questions

- Go through the distributed systems discussed throughout the course and determine where in the CAP space they lie?
- At a high-level, specify systems that are CA, CP and AP – what systems are CA?
- Is partition-tolerance a binary property? Discuss.
- See if you can specify a spectrum of C vs. A, given P.

A photograph of a server room with rows of server racks. The racks are filled with various hardware components, including hard drives and circuit boards. The room has a tiled floor and fluorescent lighting.

Distributed File Systems

Agenda

- File system basics: POSIX, ext2, etc.
- User-oriented FS: Network file systems (NFS)
- Big Data FS: GFS, (HDFS)
- Erasure coding



Distributed File Systems

File System Basics

Interaction with file systems

- POSIX – *Portable OS Interface*
 - POSIX, “*The Single UNIX Specification*”
 - Aligns with the ISO C 1999 standard (stdio.h)
 - Family of standards
- Specified by IEEE Computer Society
- Today, comprised of about 20 documents
- Abstractions for programmer to achieve platform independence (portability)
- File system interface

Basic concepts

- Files
- Directories
- Links
- Metadata
- Locks

```
chris@xr2d2 / % tree -L 1
.
├── bin
├── boot
├── cdrom
├── core
├── dev
├── etc
├── home
├── initrd.img -> boot/initrd.img-4.2.0-19-generic
├── initrd.img.old -> boot/initrd.img-4.2.0-18-generic
├── lib
├── lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
├── var
├── vmlinuz -> boot/vmlinuz-4.2.0-19-generic
└── vmlinuz.old -> boot/vmlinuz-4.2.0-18-generic

21 directories, 5 files
```


File system operations

- **File** operations:
 - Open
 - Read
 - Write
 - Close
 - ...
- **Directory** operations:
 - Create file
 - Mkdir
 - Rename file
 - Rename dir
 - Delete file
 - Delete dir

POSIX Files <stdio.h>

```
FILE *fopen(const char * filename, const char * mode)
```

Modes

```
r  open text file for reading
w  truncate to zero length or create text file for writing
a  append; open or create text file for writing at end-of-file
rb open binary file for reading
wb truncate to zero length or create binary file for writing
ab append; open or create binary file for writing at end-of-file
r+ open text file for update (reading and writing)
w+ truncate to zero length or create text file for update
a+ append; open or create text file for update, writing at end-of-file

r+b or rb+ open binary file for update (reading and writing)
w+b or wb+ truncate to zero length or create binary file for update
a+b or ab+ append; open or create binary file for update, writing at end-of-file
```

```
int fflush(FILE *stream);
//Any buffered data is physically persisted
int fclose(FILE *stream);
//File flushed and closed
```

POSIX Directories <stat.h>

```
int mkdir(const char* path, mode_t mode)

/* example
mkdir("/home/aj/distributed_systems", S_IRUSR |
    S_IWUSR | S_IXUSR | S_IRWXG );

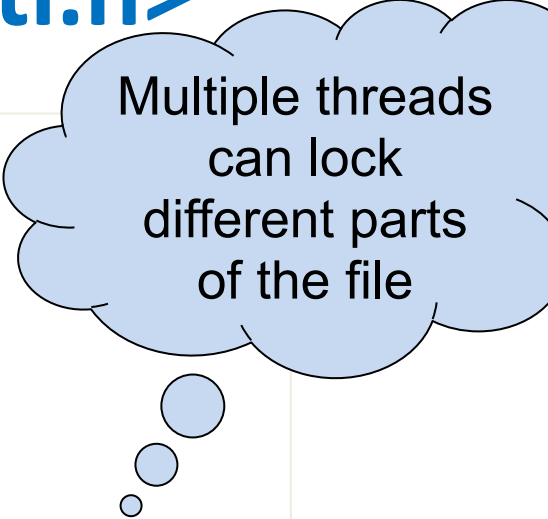
...
S_IRUSR  read permission, owner
S_IWUSR  write permission, owner
S_IXUSR  execute/search permission owner
S_IRWXG  read, write, execute/search by group
...
*/
```

POSIX File Locking <fcntl.h>

```
int fcntl(int fildes, int cmd, ...);

int fd;
struct flock fl;
fd = open("/home/aj/test.txt");
fl.l_type = F_WRLCK;           //write lock
fl.l_whence = SEEK_SET
fl.l_start = 500;              //start at byte 500
fl.l_len = 100;                //next 100 bytes

fcntl(fd, F_SETLK, &fl); //acquire lock
```



Multiple threads
can lock
different parts
of the file

Types of locks:

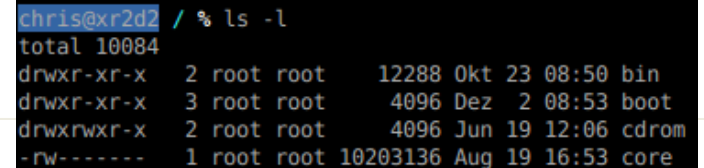
F_RDLCK	Shared or read lock
F_WRLCK	Exclusive or write lock
F_UNLCK	Unlock

POSIX File Metadata <stat.h>, <unistd.h>

```
//access permissions  
int chmod(const char * file, mode_t mode)
```

- 777 read, write, execute for all
- 664 sets **read and write** and **no execution** access for **owner** and **group**, and **read, no write, no execute** for all others

```
// set user read and write permission for file.txt  
chmod("file.txt", S_IRUSR | S_IWUSR)
```



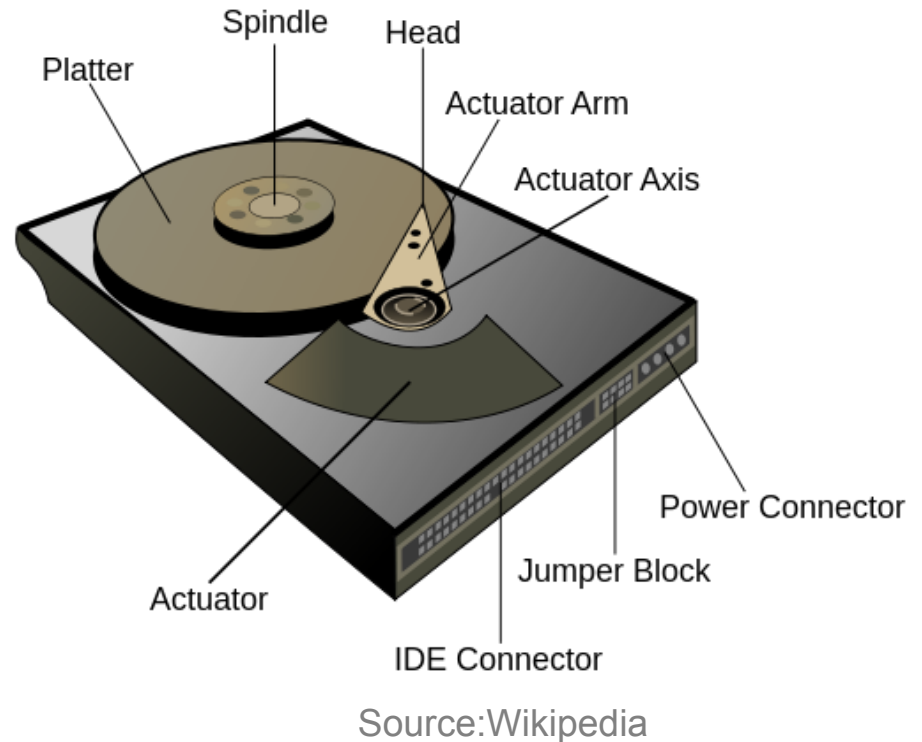
```
chris@xr2d2 / % ls -l  
total 10084  
drwxr-xr-x  2 root root   12288 Okt 23 08:50 bin  
drwxr-xr-x  3 root root    4096 Dez  2 08:53 boot  
drwxrwxr-x  2 root root    4096 Jun 19 12:06 cdrom  
-rw-----  1 root root 10203136 Aug 19 16:53 core
```

```
//change ownership of a file  
int chown(const char *, uid_t, gid_t)
```

```
//e.g. chown("file.txt", getpwnam("arno"), -1)
```

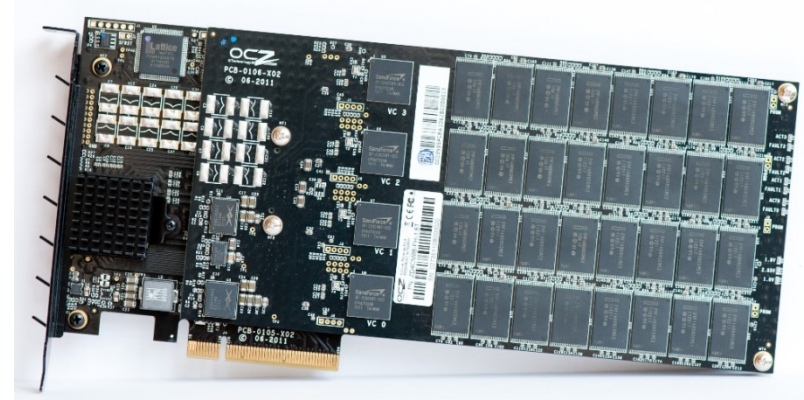
Hard-disk Drive (HDD)

- Magnetic discs
- Cache (8MB – 128MB)
- Cost
 - ~38€/TB (1.1.2014)
 - ~30€/TB (4.12.2014)
 - ~29€/TB (7.12.2015)
 - ~22€/TB (29.11.2017)
- 5400 rpm – 15000 rpm
- Seek 4-9ms
- Connected via SATA, SCSI/SAS ...



Solid-state Drive (SSD)

- DRAM (NAND-based flash memory)
- No moving mechanical components
- Cache (16MB – 512MB)
- Cost
 - ~600€/TB (1.1.2014)
 - ~350€/TB (4.12.2014)
 - ~260€/TB (7.12.2015)
 - ~250€/TB (29.11.2017)
- Can also be connected via PCI Express
- Low-level operations differ a lot compared to HDD
 - On SSD's overwriting costs more → TRIM Command
 - Deleting is delegated to internal firmware which has a garbage collector



Source:OCZ

How common are HDD failures?

Backblaze Hard Drive Failure Rates

Ordered by Drive Size (2013 through Q3 2015)

Model Name/Number	Size	2013 Failure Rate	2014 Failure Rate	2015 Failure Rate	Failure Rate	All Periods: 2013-2015 Low Rate	High Rate
All 1.5TB Drives		16.57%	13.11%	15.10%	14.71%		
HGST(*) Deskstar 7K2000 (HDS722020ALA330)	2TB	1.03%	1.07%	2.81%	1.61%	1.40%	1.90%
Seagate Barracuda LP (ST32000542AS)	2TB	7.90%	13.43%		10.28%	6.90%	14.20%
Western Digital Red (WD20EFRX)	2TB		0.00%	6.85%	6.85%	2.40%	17.50%
All 2TB Drives		1.45%	1.42%	2.87%	1.88%		
HGST(*) Deskstar 5K3000 (HDS5C3030ALA630)	3TB	0.99%	0.59%	1.31%	0.92%	0.70%	1.10%
HGST(*) Deskstar 7K3000 (HDS723030ALA640)	3TB	1.01%	2.27%	2.12%	1.91%	1.40%	2.60%
Seagate Barracuda 7200.14 (ST3000DM001)	3TB	10.35%	43.08%	30.94%	28.46%	26.90%	29.60%
Seagate Barracuda XT (ST33000651AS)	3TB	6.91%	4.80%	3.55%	5.11%	3.50%	7.30%
Toshiba DT01ACA Series (TOSHIBA DT01ACA300)	3TB	6.93%	3.68%	2.80%	4.20%	1.40%	9.80%
Western Digital Red 3 TB (WDC WD30EFRX)	3TB	3.79%	6.94%	8.79%	7.65%	6.40%	9.30%
Western Digital Green 3 TB (WDC WD30EZRX)	3TB	6.32%	0.00%		6.32%	4.10%	9.80%
All 3TB Drives		5.22%	15.06%	4.33%	9.43%		

Hard Drive Failure Rates. All Drives. All Manufacturers.

(Year 2013-2015)



~5% fail per year

~10% failed during
first 3 years

Bitrot on HDD

- Bitrot means silent corruption of data
- HDD specifications predict an **Uncorrectable bit Error Rate** (UER) of 10^{15} (1,000,000,000,000,000 ~ 125 TB)
- Evaluation [1]
 - 8x100GB HDD
 - After 2 PB reads
 - 4 read errors where observed
- How to protect against bitrot?
 - Erasure codes

[1] <http://research.microsoft.com/pubs/64599/tr-2005-166.pdf>

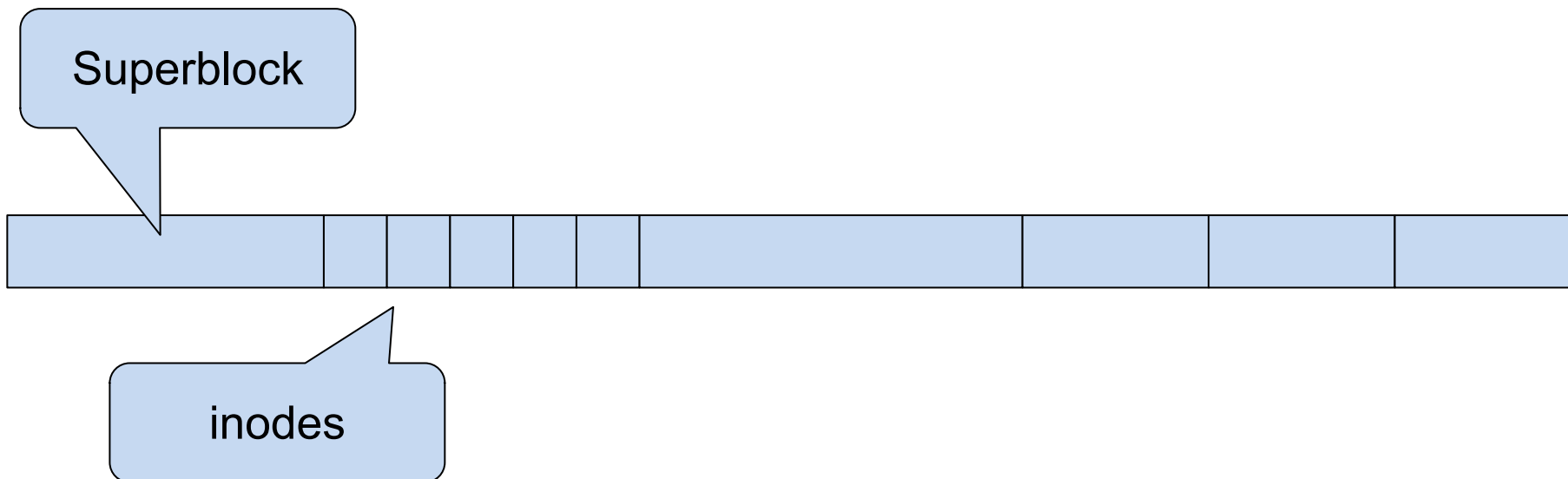
Disk file systems

- Linux
 - ext, ext2, ext3, ext4
 - JFS, XFS,
 - BTRFS, ZFS
 - Pooling, snapshots, checksums
- Windows
 - NTFS
 - FAT, FAT32, exFAT, ReFS
- Let's take a quick look at ext2

Linux ext2

The second extended file system

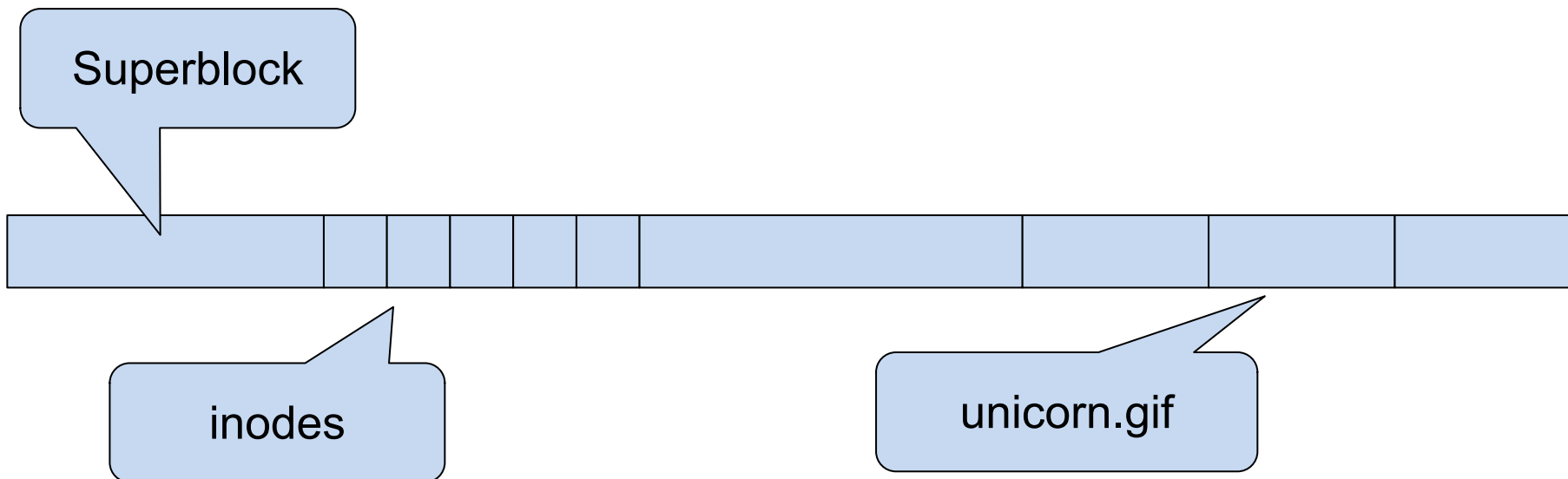
- **Superblock**, file system metadata (repeated)
 - Defines file system type, size, status, and information about other metadata structures (metadata of metadata)
- Index-nodes (**inodes**): one per file or directory
- **Data blocks**



Linux ext2

The second extended file system

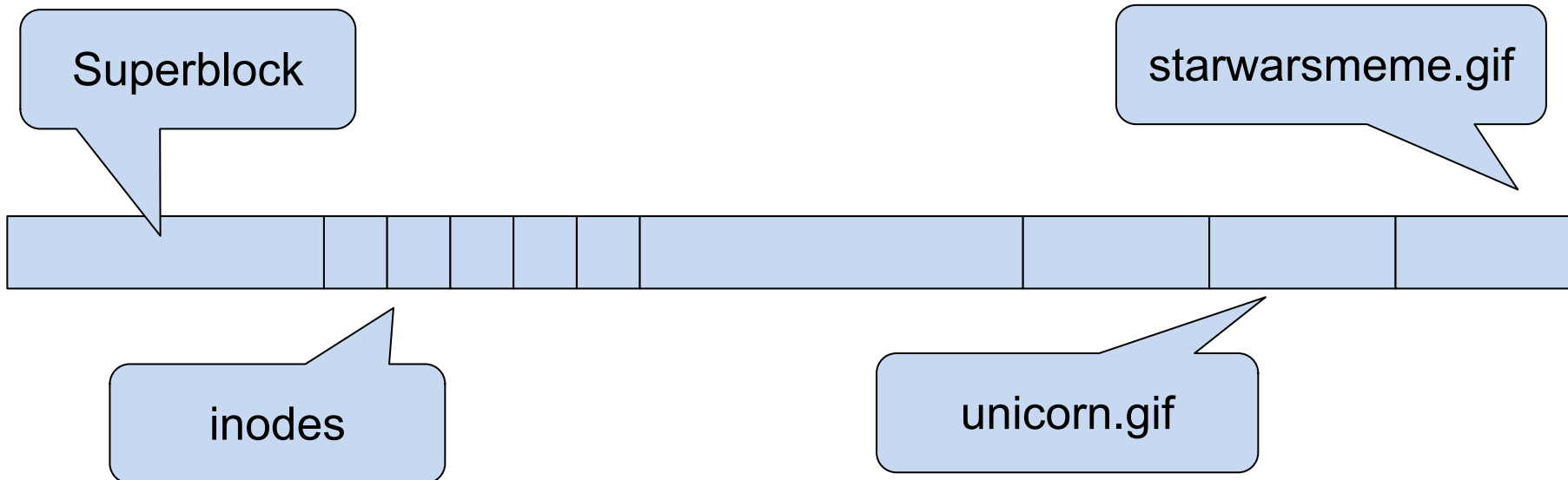
- **Superblock**, file system metadata (repeated)
 - Defines file system type, size, status, and information about other metadata structures (metadata of metadata)
- Index-nodes (**inodes**): one per file or directory
- **Data blocks**



Linux ext2

The second extended file system

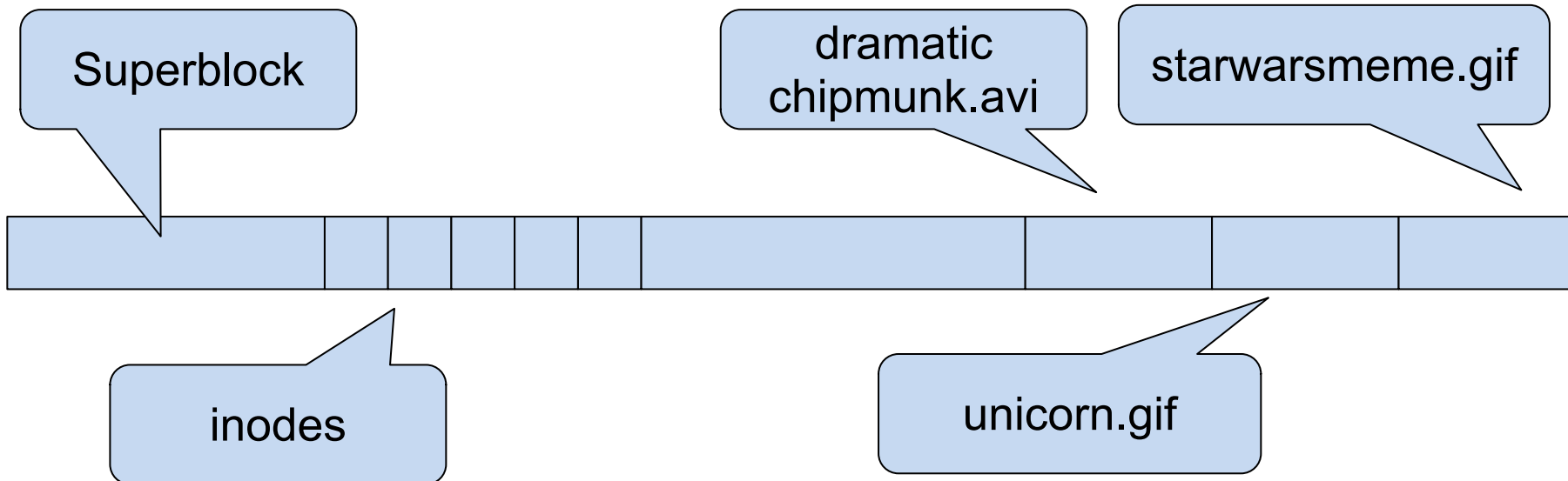
- **Superblock**, file system metadata (repeated)
 - Defines file system type, size, status, and information about other metadata structures (metadata of metadata)
- Index-nodes (**inodes**): one per file or directory
- **Data blocks**



Linux ext2

The second extended file system

- **Superblock**, file system metadata (repeated)
 - Defines file system type, size, status, and information about other metadata structures (metadata of metadata)
- Index-nodes (**inodes**): one per file or directory
- **Data blocks**



ext2 inode

- Owner and group identifiers
- File length
- File type and access rights
- Number of data blocks
- Array of pointers to data blocks
- Timestamp

- Types
 - File
 - Directory
 - Symbolic link

Linux/fs/ext2/ext2.h

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
    __le16 i_mode;    /* File mode */
    __le16 i_uid;     /* Low 16 bits of Owner Uid */
    __le32 i_size;    /* Size in bytes */
    __le32 i_atime;   /* Access time */
    __le32 i_ctime;   /* Creation time */
    __le32 i_mtime;   /* Modification time */
    __le32 i_dtime;   /* Deletion Time */
    __le16 i_gid;     /* Low 16 bits of Group Id */
    __le16 i_links_count; /* Links count */
    __le32 i_blocks;  /* Blocks count */
    __le32 i_flags;    /* File flags */
    ...

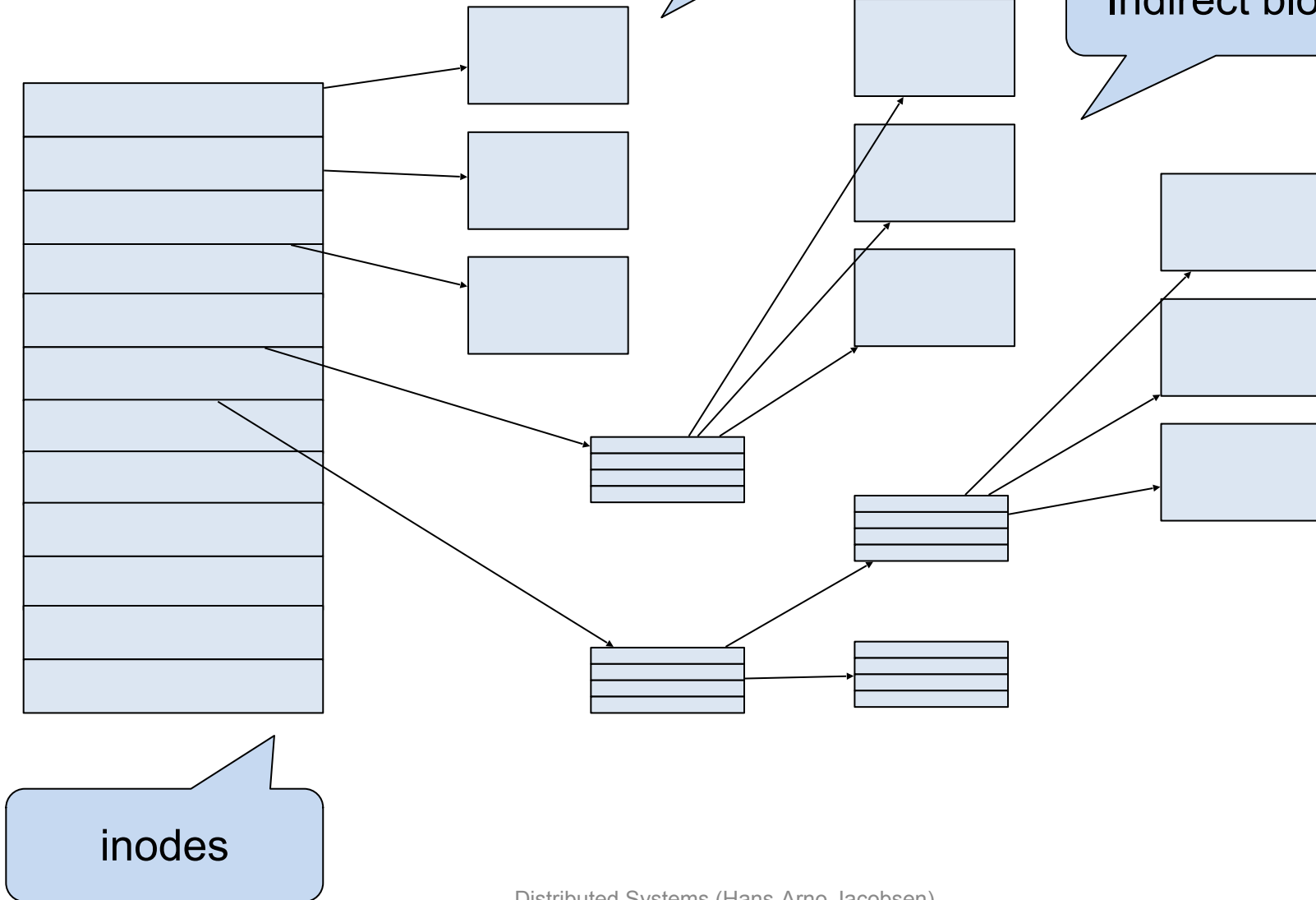
    struct ext2_dir_entry {
        __le32 inode;    /* Inode number */
        __le16 rec_len;  /* Directory entry length */
        __le16 name_len; /* Name length */
        char name[];     /* File name, up to EXT2_NAME_LEN */
    };
};
```

Source: <https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h>

ext2 inode

Direct blocks

Indirect blocks





Distributed File Systems

NFS – Network File System

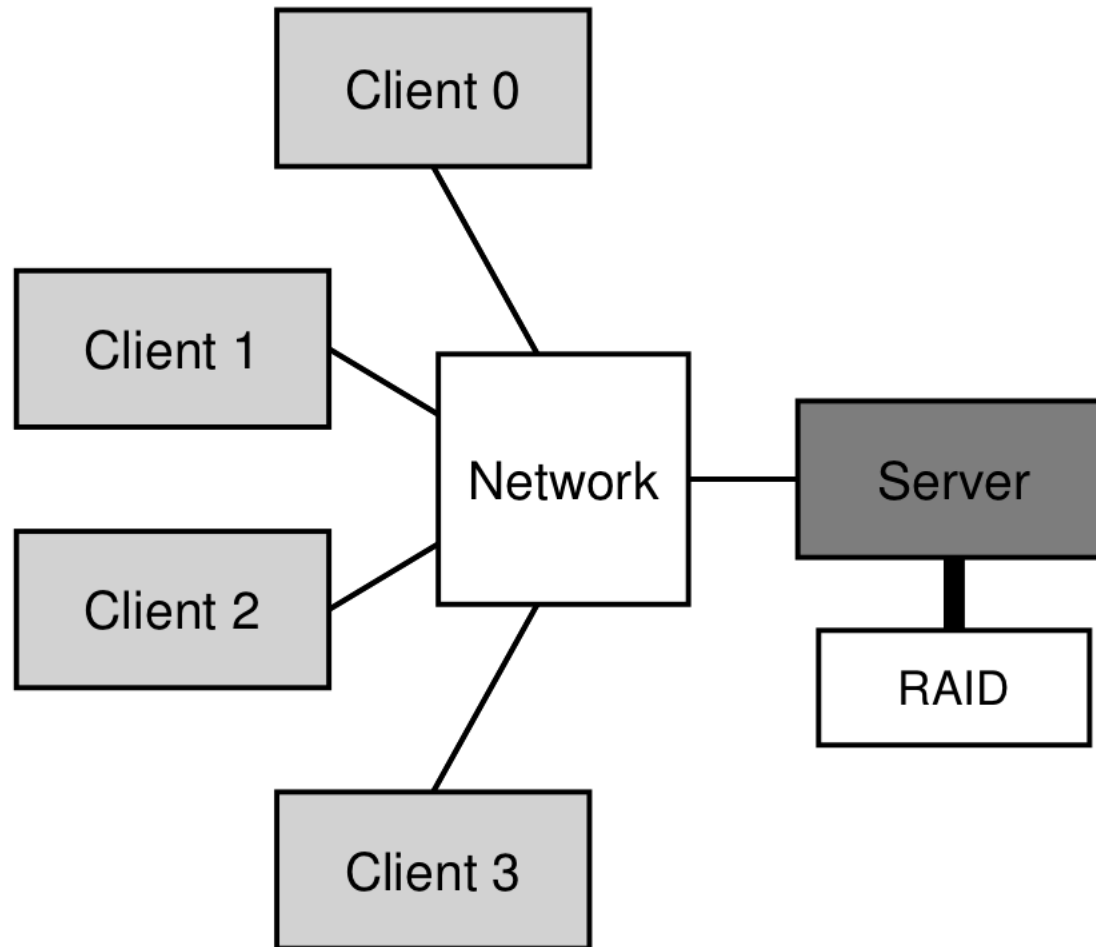
Distributed file systems

Motivation

- Collaboration
 - Shared file directory for projects, etc.
- Resource sharing
 - Pooling resources accross multiple devices
 - Incremental scalability (add hardware over time)
- Challenges
 - Performance
 - Scalability
 - Consistency

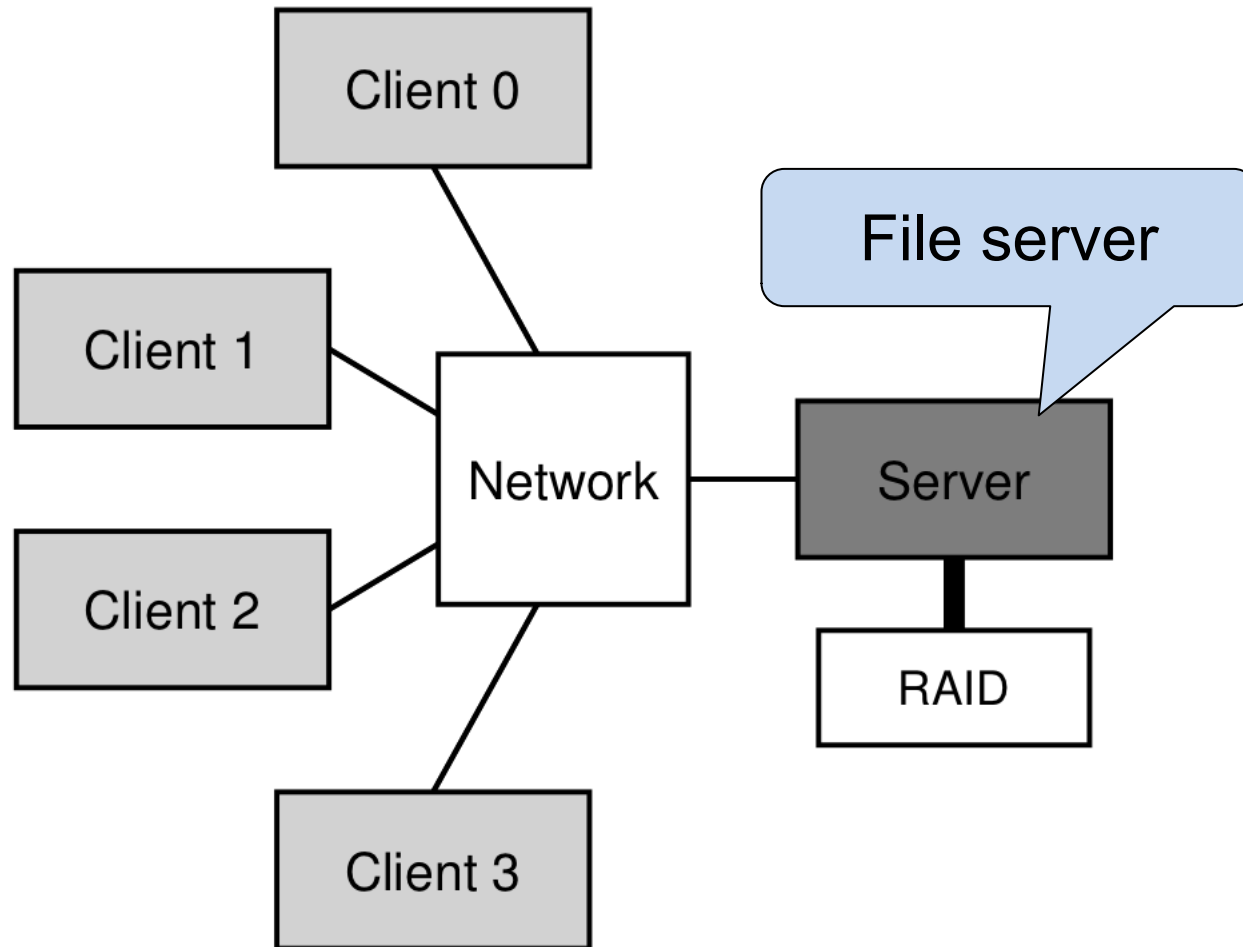
Distributed file system

Simplified



Distributed file system

Simplified



The Network File System (NFS)

Initially, 1984, by Sun Microsystems

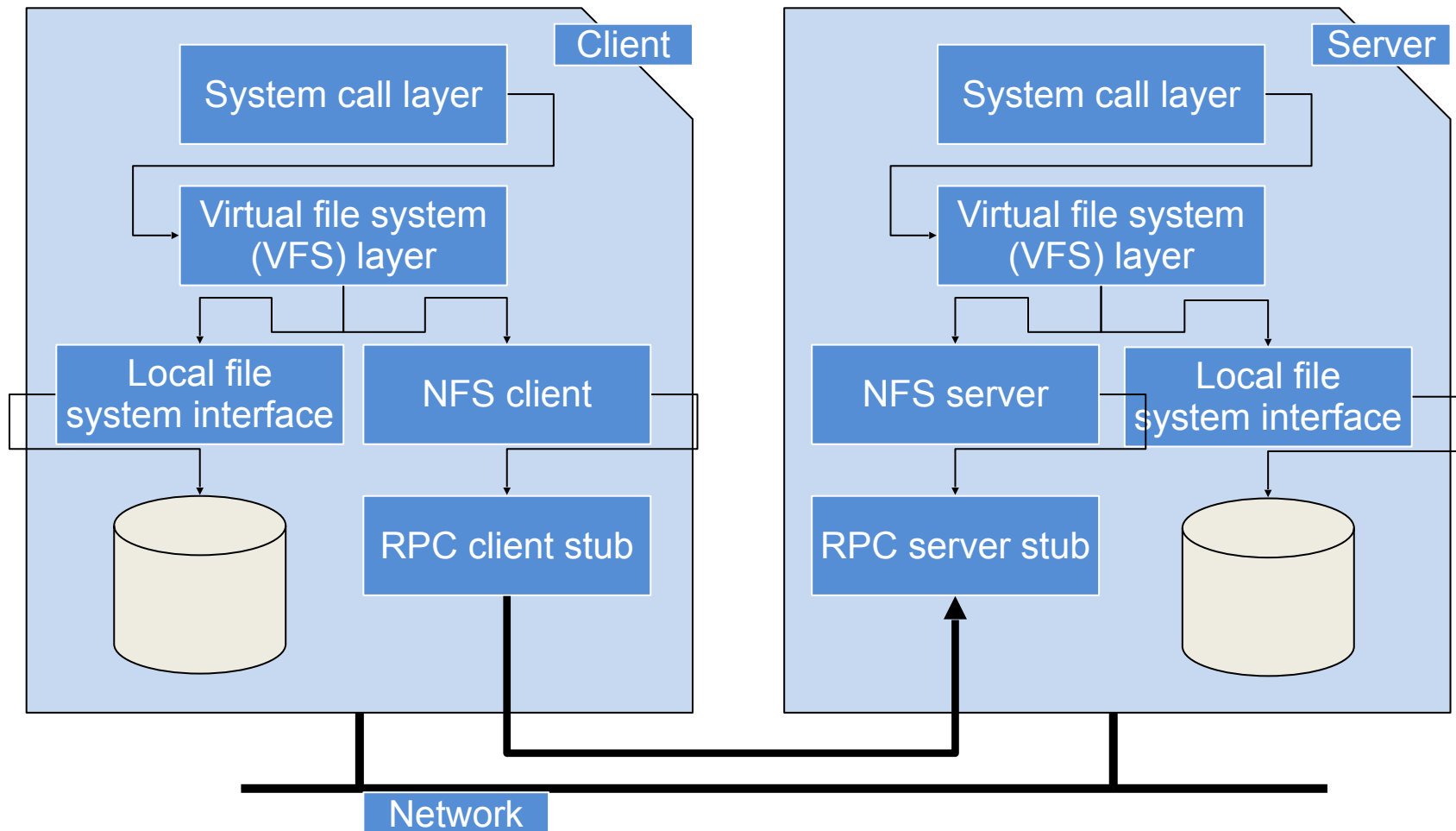
- Goals:
 - **Consistent namespace** for files across nodes
 - ***Authorized users*** can access their files from any node
- NFS protocol designed for LANs
- NFS creates a remote access layer for file systems
 - Each file is hosted on a **server** and accessed by **clients**
 - Namespace is **distributed** across servers
 - Each client treats remote files as local ones (“virtual files”)

The Network File System (NFS)

Initially, 1984, by Sun Microsystems

- NFS follows a **user-centric** design
 - Most files are privately owned by a **single user**
 - Few **concurrent access** across clients
 - Reads are **more common** than writes
- Open protocol
 - Lead to wide adoption
 - Many commercial implementation

Basic NFS architecture



Sending commands

- Essentially, NFS works as a replicated system using **remote procedure calls (RPCs)** to propagate FS operations from client(s) to server(s)
- Naïve solution: forward **every RPC** to server
 - Server orders all incoming operations, performs them, returns results
- Downside
 - High access latency due to RPCs
 - Server becomes overloaded by many RPCs

Solution: Caching

- Clients use a cache to store a copy of remote files, called “virtual files”
- Clients periodically synchronize with server
- This is essentially **multi-primary replication**:
 - *How should synchronization be done? (eager/lazy)*
 - *What is the right consistency level?*

Original version: Sun NFS

NFSv2, ..., NFSv4, ...

- Developed in 1984
- Uses in-memory caching:
 - File blocks, directory metadata
 - Stored at both clients and servers
- Advantage: no network traffic for open, read, write
- Problems: **failures** and **cache consistency**

Failures I

- **Server crash**
- Any data not persisted to disk is lost
- What if client does seek(); [***server crash***]; read()?
 - Seek sets a position offset in the opened file
 - After crash, **server forgets** offset, **read returns incorrect data**

Failures II

- **Communication omission failures**
 - Client *A* sends `delete(foo)`, server processes it
 - Server acknowledgement of delete is lost, meanwhile Client *B* issues `create(foo)`
 - Client *A* times out and send `delete(foo)` again, **deleting the file created by Client *B*!**
- **Client crash**
 - Since caching is in memory, **lose all updates by client** if not synched to server

Solution: Stateless RPC

- RPC commands are **stateless**: server does not maintain state across commands in a “session”
- *read()* is stateful (server needs to remember *seek()*) → **read(position)** is **stateless** (server has all the information needed for correct read)
- With stateless RPC, server can fail and later continue to serve commands without recovering former state

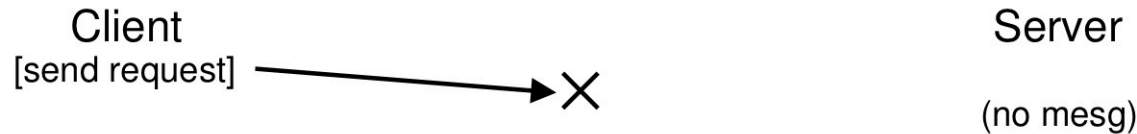
Solution: Idempotent RPC

- NFS's RPCs are designed to be **idempotent**
- Repeating a command has no side effect
- `Delete("foo")` becomes **`delete(someid)`**, so it cannot wrongly delete a new file named "foo"
- Read, lookup are idempotent
- Write includes, data to write, the file ID, the offset to write at, therefore, idempotent

Common loss scenarios

Handled by client via timeout, retry, idempotent server RPCs

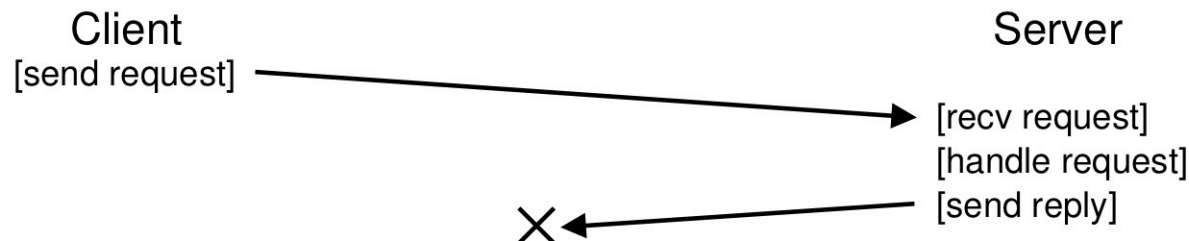
Case 1: Request Lost



Case 2: Server Down



Case 3: Reply lost on way back from Server



Is mkdir idempotent?

- MKDIR message from client to server
- Server ACKing successful creation is lost
- Client times out and retries MKDIR
- Server responds with error directory exists
- NFS designers opted to keep design simple

Cache Consistency

- Clients can cache file blocks, directory metadata, *etc.*



- *What happens if both clients want to write?*

Solution: Time-bounded consistency

- Flush-on-close: When a file is **closed**, modified blocks are sent to server **synchronously** (close() does not return until update is finished)
- Each client periodically checks with server for updates
- Clients synchronize their cache **after some bounded time** if there are no more updates; otherwise they would read stale data

Concurrent Writes in NFS

- NFS does not provide any guarantees for concurrent writes!
- Server may update using one client's writes, other's writes, or a mix of both!
- Not usually a concern due to the **user-centric** design: assuming there are no concurrent writes
- A big problem if one needs to support concurrent writes

NFS Summary

- **Transparent** remote file access
- **Client-side caching** for improved performance
- **Stateless** and **idempotent** RPCs for fault-tolerance
- Periodical synchronization with the server, with **flush-on-close semantics**
- No guarantees for concurrent writes



Distributed File Systems

GFS – Google File System

The Google File System (GFS)

Design assumptions

- Designed for Big Data workloads
 - Huge files (100MB+), not optimized for small files
- Fault tolerance while running on inexpensive commodity hardware
 - 1000s machines where failure is the norm
- Introduces an API which is designed to be implemented scalably (non-POSIX)
- Architecture: one master, many chunk (data) servers; can operate across WAN links
 - Master stores metadata and monitors chunkservers
 - Chunkservers store and serve data chunks

Design assumptions

- Read workload
 - Large streaming reads (data caching not beneficial); no client-side data caches
 - Small random reads
- Write workload
 - File append via producer-consumer pattern
 - Hundreds of concurrently appending clients
 - Modification supported but not a design goal
- Bandwidth is more important than low latency

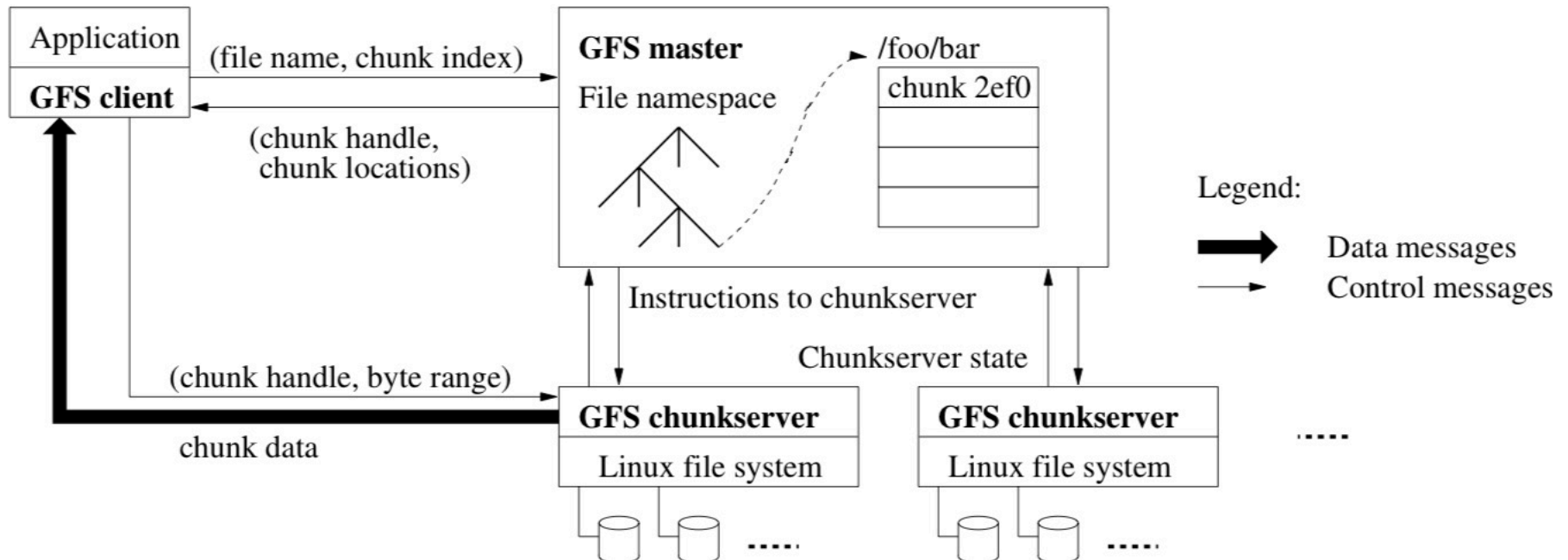
Interface

- Supported operations
 - Create, delete, open, close, read, write
 - Record append
 - Allows multiple clients to append data to the same file while guaranteeing atomicity
 - Snapshot
 - Creates a copy of a file or a directory tree at low costs
- Does not support full POSIX interface
 - POSIX requires many guarantees which are hard to fulfill in distributed applications

Architecture

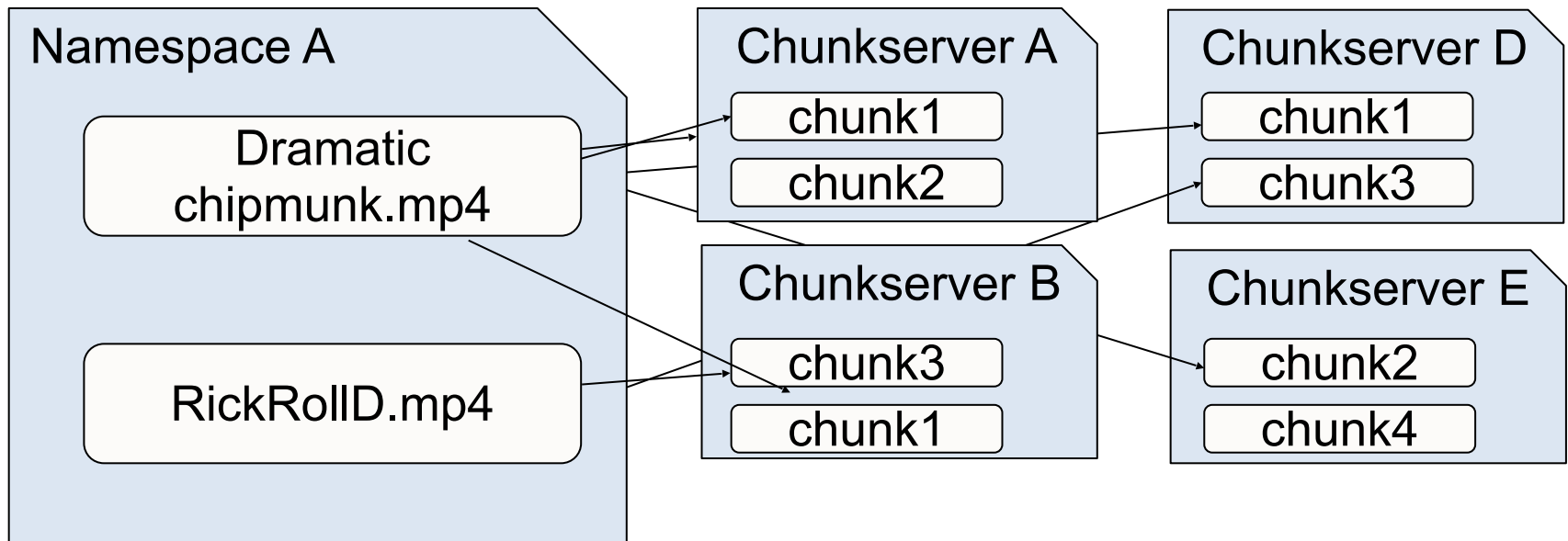
- Files
 - Divided into fixed-size chunks (64MB)
 - Identified by an immutable and unique id (chunk handle)
- Single master
 - Maintains GFS metadata
 - Namespace, access control information, mapping from files to chunks, location of chunks
 - Heartbeats chunkservers
- Multiple chunkservers
 - Chunks are stored on disk as Linux files
 - Each chunk is replicated to multiple chunkservers (depending on a replication factor defaulting to 3)

GFS architecture



Metadata kept at Master

- File and chunk namespaces
- Mapping from files to chunks
- Location of each chunk's replicas



Metadata management by Master

- Replicated to a shadow master and logged to operation log
 - **Namespaces**
 - **File to chunk mapping**
- **Location of chunks** is in-memory only (fast)
 - At start-up, periodically, upon failover, master asks chunkservers which chunks they have to rebuild location-to-chunk mapping
 - Periodic scanning is used to implement
 - Garbage collection (when files are deleted)
 - Re-replication (chunkserver failure)
 - Chunk migration (to balance load and disk space)
- Metadata has to fit in memory (64 bytes/chunk)

Operation log at Master

- Maintains all file creating, renaming, deletion operations *etc.*
- Only persistent, historical record of metadata changes
- Persisted to local disk and replicated to shadow master(s)
- Metadata changes are only visible after they are persisted
- Serves for Master recovery by replaying operation log
- Periodic checkpointing of Master state to minimize replaying effort

How is fault-tolerance achieved?

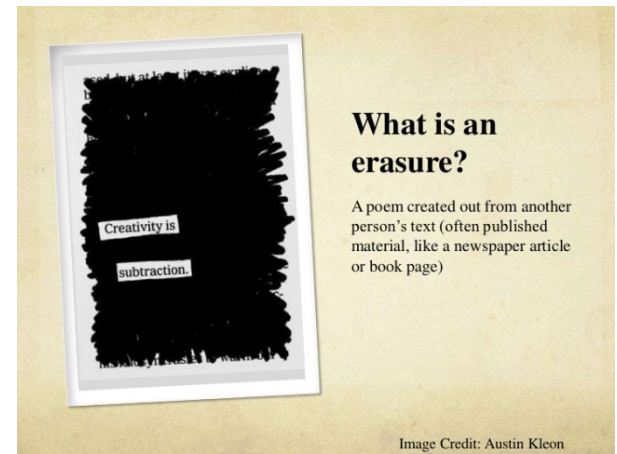
- Master
 - Operation log, replication of log and metadata to shadow master
- Chunkserver
 - All chunks are versioned
 - Version number updated upon modification
 - Chunks with old versions are not served and are deleted
- Chunks
 - Re-replication triggered by master, maintains replication factor
 - Rebalancing to distribute load among chunkservers
 - Data integrity checks

How is high-availability achieved?

- Fast recovery of master
 - Checkpointing and operation log
- Shadow master(s)
 - Serve read traffic, reduces downtime during failover
- Heartbeat messages (often include piggy-backed status updates)
 - Discover chunkserver failure
 - Trigger re-replication
 - Share current load
 - Trigger garbage collection
- Diagnostic tools

Summary on GFS

- Highly concurrent reads and appends
- Highly scalable
- On cheap commodity hardware
- Built for map-reduce kind of workloads
 - Reads
 - Appends
- Developers have to understand the limitations and may have to use other mechanisms to work around
- No POSIX API, would require many guarantees which are difficult to fulfill in DS



ERASURE CODES

INTRODUCTION

Erasures

- Byte errors where we ***know*** the **position** of the **dropped** or corrupted **bytes** is called an ***erasure***
- As opposed to byte errors where we don't know the position of the dropped or corrupted bytes

Types of Erasure Codes

- **Linear block code**
 - Reed Solomon Code
 - Can sustain lost bytes of known position
 - Distributed file systems
- **Fountain code**
 - LT Codes (Luby Transform)
 - Can sustain lost bytes of unknown position
 - P2P systems, torrents, video streaming ...

Reed Solomon Code

- Error-correcting code
 - Used in QR codes
- Block encoding
 - **Data blocks**
 - Error-correcting blocks (**parity blocks**)
 - Read data blocks first
 - Else, decode with parity blocks

General Problem

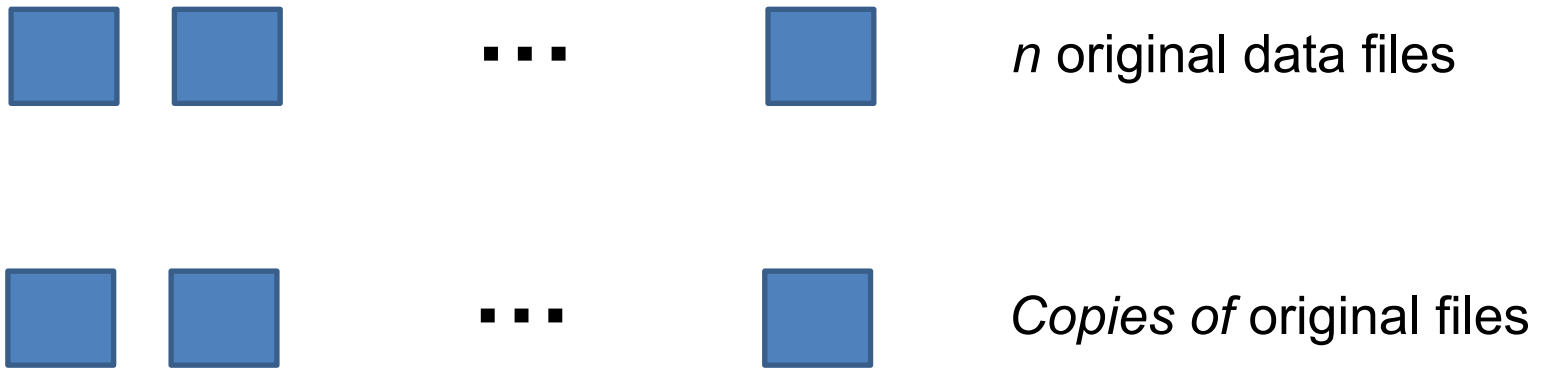
(n, m) -code



- We have **n data files**, guard against **losing m** of them
- Generate **m parity files**
- Lose up to m data files, can use equal number of parity files to recover data files
- Also works if some parity files are lost, as long as there are **n files left** (parity or data), can recover original data files
- Compare creating n parity files to making *a* copy of n data files (a.k.a. common backup strategy)

Recoverability

Via backup strategy

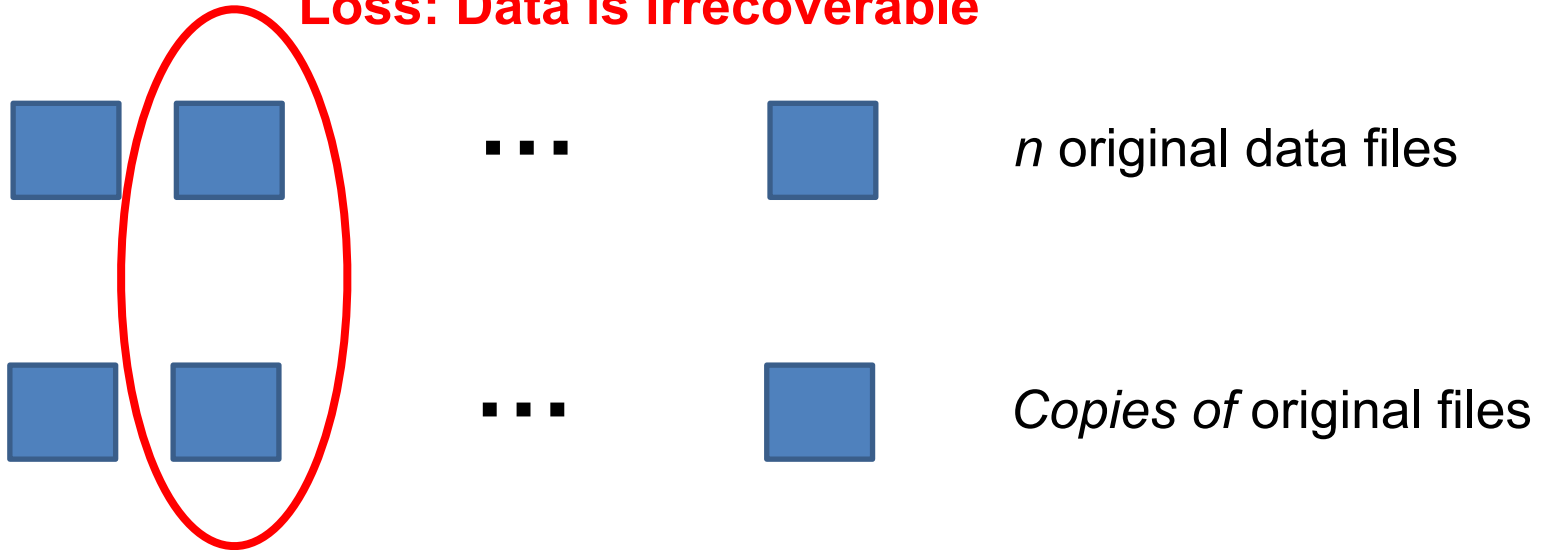


- With n original files and n copies (one each), if we lose two files (original and its copy), we can't recover loss
- Total storage requirement is $2n$

Recoverability

Via backup strategy

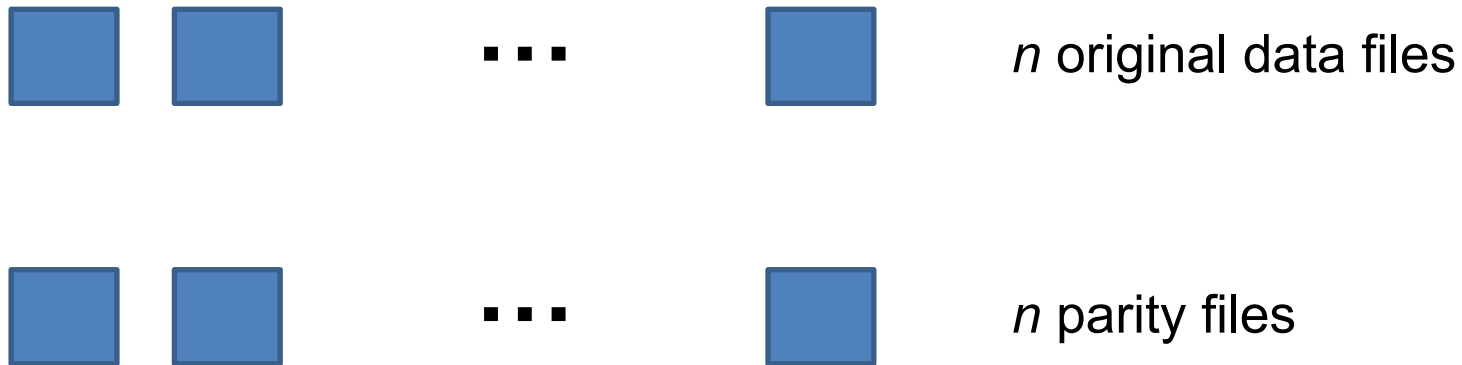
Loss: Data is irrecoverable



- With n original files and n copies (one each), if we loose two files (original and its copy), we can't recover loss
- Total storage requirement is $2n$

Recoverability

Via an (n, m) -code



- With an (n, m) -code, we can protect n data files against the loss of m of them by generating m parity files
- Say, we use an (n, n) -code with total storage requirement $2n$
- Could lose up to n files (any combination of data and parity files)

Recoverability

Via an (n, m) -code

Loss: Data remains recoverable



- With an (n, m) -code, we can protect n data files against the loss of m of them by generating m parity files
- Say, we use an (n, n) -code with total storage requirement $2n$
- Could lose up to n files (any combination of data and parity files)

Recoverability

Via an (n, m) -code

Parity files take **same amount of space** but provide **superior (recovery capabilities!)**

Loss: Data remains recoverable



- With an (n, m) -code, we can protect n data files against the loss of m of them by generating m parity files
- Say, we use an (n, n) -code with total storage requirement $2n$
- Could lose up to n files (any combination of data and parity files)

Optimal Erasure Code

- Above sketched (n, m) -code is an erasure code because it guards against byte erasures
- It does not guard against more general errors where we don't know which data bytes have been corrupted
- Called optimal because in general, we **need at least n known bytes to recover n data bytes**, bound achieved here

Erasure Code Computations

Parity & Data Reconstruction Computation

- Given a pair (n, m) and n data bytes
- **Compute parity data:** compute m parity bytes, given n data bytes
- **Reconstruct data:** given a partial list of at least n data and parity bytes
 - Return full list of data bytes, i.e., there are no more than m omitted data or parity bytes
 - Error otherwise
- Generally, operates on byte level

Erasure Code

$n, m = 1$

- For d_i a data byte, compute parity byte p
 - $p = d_0 + d_1 + \dots + d_{n-1}$
- For $m = 1$, guard against loss of a single d_i
- Reconstruct data
 - $d_i = p - (d_0 + d_1 + d_{i-1} + d_{i+1} + \dots + d_{n-1})$

Nota Bene

Above Erasure Code with $n, m=1$

- Based on **bytes** in practice
- Computation is done via **modulo 256 arithmetic**
- Addition and subtraction wrap around
- Multiplication and division are not required



Pixabay.com

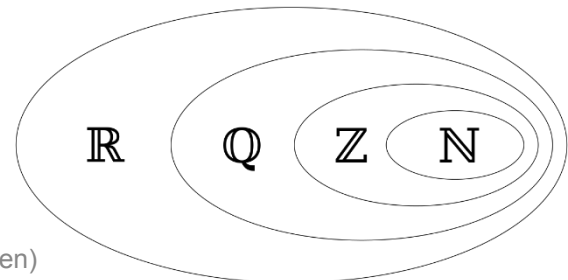
EC Computations

More Generally Speaking

- Based on **bytes** in practice
- Computations done over **fields** (e.g., Galois Fields) where
 - addition and subtraction
 - multiplication and division
 - existence of additive inverse $-a$ for all elements a , and of a multiplicative inverse b^{-1} for every nonzero element b

are defined (i.e., computation over bytes with Galois Field)

- For simplicity and illustration, here, we use the **Rationales \mathbb{Q}** in our calculations (\mathbb{Q} is a **field** as well so all operations we need are defined)
- Including in assignments, *etc.*, unless explicitly stated otherwise!

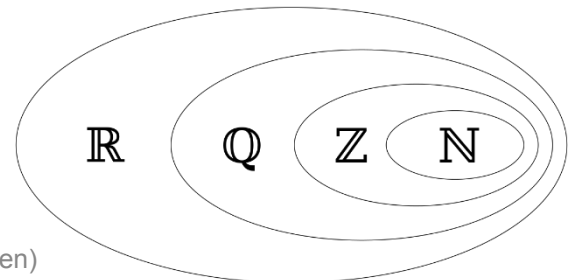


EC Computations

More Generally Speaking

$$\frac{1}{8} \quad \frac{1}{2}$$
$$\frac{1}{4} \quad \frac{1}{3}$$

- Based on **bytes** in practice
 - Computations done over **fields** (e.g., Galois Fields) where
 - addition and subtraction
 - multiplication and division
 - existence of additive inverse $-a$ for all elements a , and of a multiplicative inverse b^{-1} for every nonzero element b
- are defined (i.e., computation over bytes with Galois Field)
- For simplicity and illustration, here, we use the **Rationales \mathbb{Q}** in our calculations (\mathbb{Q} is a ***field*** as well so all operations we need are defined)
 - Including in assignments, *etc.*, unless explicitly stated otherwise!



Erasure Code

$$n = 3, m = 2$$

- (3, 2)-code
- Here, p_i **parity bytes**, d_i **data bytes** (i.e., numbers)
- Parity byte equations must be “*sufficiently different*”

$$p_0 = d_0 + d_1 + d_2$$

$$p_1 = 1 * d_0 + 2 * d_1 + 3 * d_2$$

- For example, the following is not “sufficiently different”

$$p_1 = 2 * d_0 + 2 * d_1 + 2 * d_2$$

- Here, $p_1 = 2 p_0$ (**avoid linear combinations**)

For two **missing data bytes**, d_i, d_j $i < j$, and **given parity bytes** p_0, p_1 , we rearrange parity equations to move unknown (i.e., missing data bytes) to left hand side:

Given parity
equations:

$$p_0 = d_0 + d_1 + d_2$$

$$p_1 = 1 * d_0 + 2 * d_1 + 3 * d_2$$

Rearranged (to solve for
missing data bytes):

$$d_i + d_j = X = p_0 - d_k$$

$$(i + 1) * d_i + (j + 1) * d_j = Y = p_1 - (k + 1) * d_k$$

where d_k is the **known (not missing) data byte**

For two **missing data bytes**, d_i, d_j $i < j$, and **given parity bytes** p_0, p_1 , we rearrange parity equations to move unknown (i.e., missing data bytes) to left hand side:

Given parity
equations:

$$p_0 = d_0 + d_1 + d_2$$

$$p_1 = 1 * d_0 + 2 * d_1 + 3 * d_2$$

Rearranged (to solve for
missing data bytes):

$$d_i + d_j = X = p_0 - d_k$$

$$(i + 1) * d_i + (j + 1) * d_j = Y = p_1 - (k + 1) * d_k$$

Equations kept generic since
we don't know upfront which
bytes get lost

where d_k is the **known (not missing) data byte**

For two **missing data bytes**, d_i, d_j $i < j$, and **given parity bytes** p_0, p_1 , we rearrange parity equations to move unknown (i.e., missing data bytes) to left hand side:

Given parity
equations:

$$p_0 = d_0 + d_1 + d_2$$

$$p_1 = 1 * d_0 + 2 * d_1 + 3 * d_2$$

Rearranged (to solve for
missing data bytes):

$$d_i + d_j = X = p_0 - d_k$$

$$(i + 1) * d_i + (j + 1) * d_j = Y = p_1 - (k + 1) * d_k$$

where d_k is the **known (not missing) data byte**

Multiply first equation by $(i+1)$ and subtract it from
second one to cancel the d_i term;
then, use first equation to solve for d_i

$$d_i + d_j = X = p_0 - d_k$$

$$(i + 1) * d_i + (j + 1) * d_j = Y = p_1 - (k + 1) * d_k$$

$$d_j = (Y - (i + 1) * X) / (j - i)$$

$$d_i = X - d_j = ((j + 1) * X - Y) / (j - i)$$

We now have equations for d_i , d_j , $i < j$ to reconstruct the
missing data from known parity bytes.

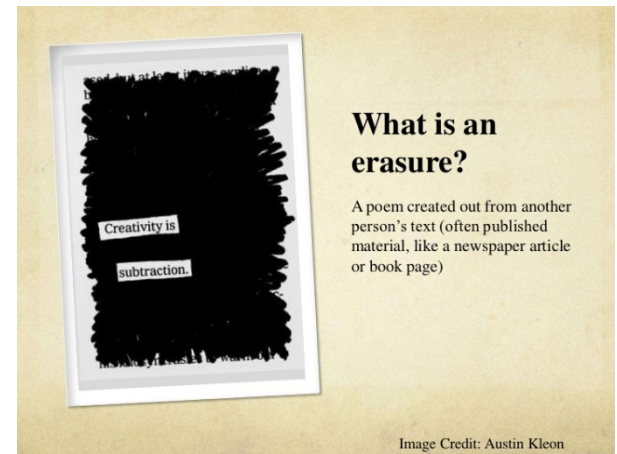
Multiply first equation by $(i+1)$ and subtract it from second one to cancel the d_i term;
then, use first equation to solve for d_i

$$\begin{aligned}d_i + d_j &= X = p_0 - d_k \\(i + 1) * d_i + (j + 1) * d_j &= Y = p_1 - (k + 1) * d_k\end{aligned}$$

$$d_j = (Y - (i + 1) * X) / (j - i)$$

$$d_i = X - d_j = ((j + 1) * X - Y) / (j - i)$$

We now have equations for $d_i, d_j, i < j$ to reconstruct the missing data from known parity bytes.



ERASURE CODES

BASIC LINEAR ALGEBRA

Basic Linear Algebra

Digression

Equations of parity numbers from above example are of the form:

$$p = a_0 * d_0 + a_1 * d_1 + a_2 * d_2$$

where a_i are constants

These are **linear combinations** of d_i s and written as

$$p = (a_0 \quad a_1 \quad a_2) \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Correspondence

Linear Functions and Matrices: Taking n inputs to m outputs

Two parity numbers, each a linear combination of d_i s:

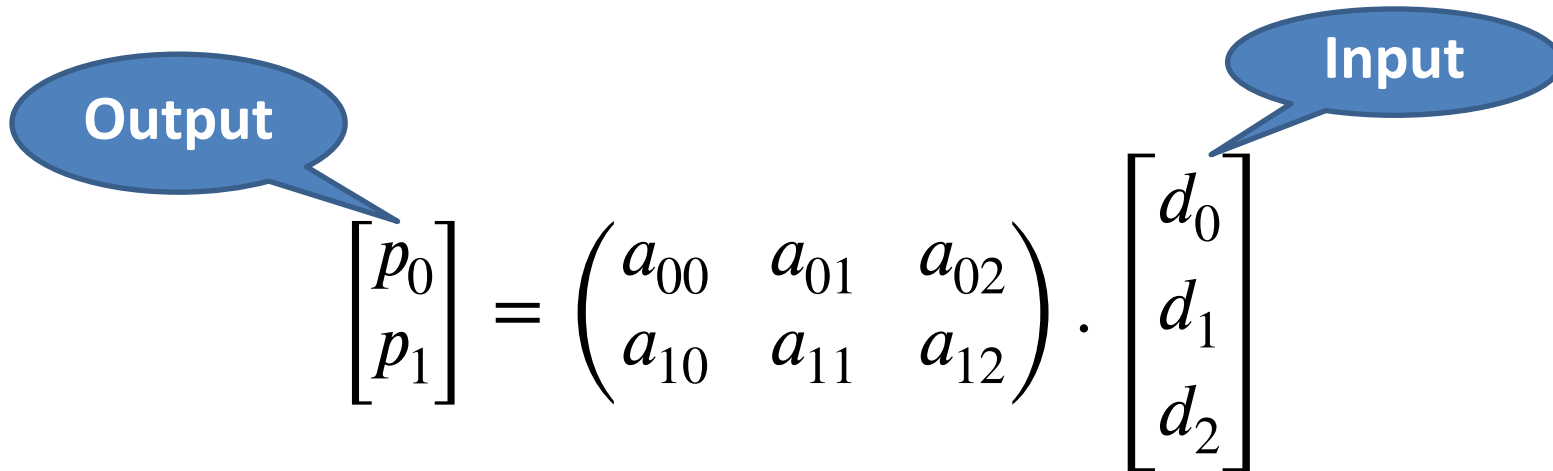
$$p_0 = a_{00} * d_0 + a_{01} * d_1 + a_{02} * d_2$$

$$p_1 = a_{10} * d_0 + a_{11} * d_1 + a_{12} * d_2$$

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear Function vs. Matrix

- *Deleting a row of a matrix corresponds to deleting an output of a linear function*



The diagram illustrates a linear transformation. On the left, a blue speech bubble labeled "Output" points to a column vector $\begin{bmatrix} p_0 \\ p_1 \end{bmatrix}$. In the center is a 2x3 matrix $\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}$. On the right, a blue speech bubble labeled "Input" points to a column vector $\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$. The equation is written as $\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$.

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

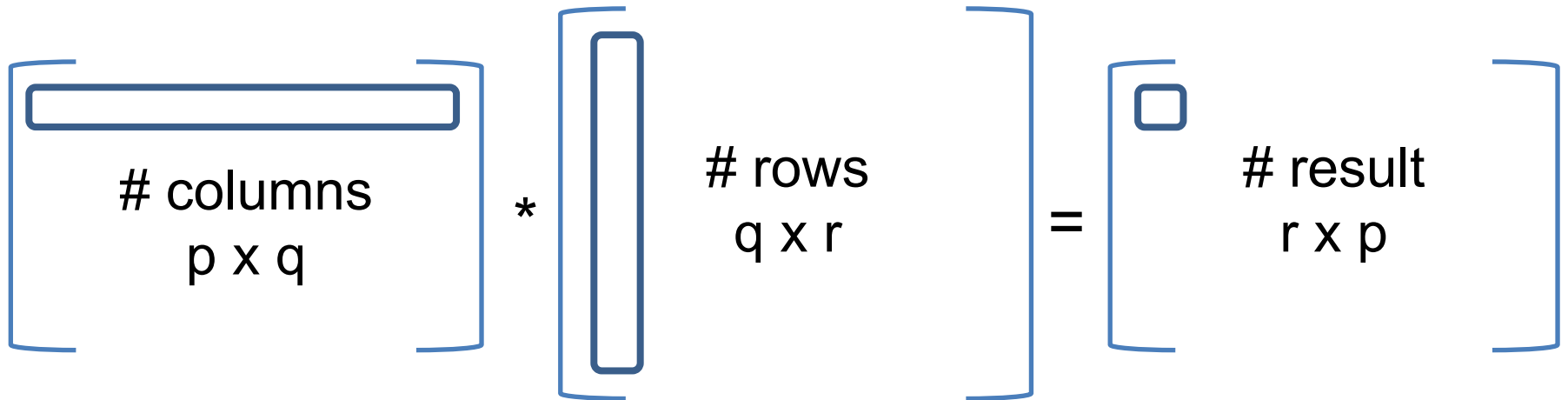
Linear Function vs. Matrix

- *Deleting a row of a matrix corresponds to deleting an output of a linear function*
- *Swapping two rows of a matrix corresponds to swapping two outputs of a linear function*
- *Appending a row to a matrix corresponds to adding an output to a linear function*

Matrix Multiplication

- Multiply matrices A , B , if they are **compatible**
- Number of columns of A must equal number of rows of B
 - A is $\mathbf{p \times q}$ matrix (#rows x #columns)
 - B is a $\mathbf{q \times r}$ matrix
 - Resulting C is an $\mathbf{r \times p}$ matrix

Matrix Multiplication



Matrix Multiplication

MATRIX-MULTIPLY (*A*, *B*)

1. **if** *A.columns* \neq *B.rows*
2. **error** "incomplete dimensions"
3. **else** let *C* be a new *A.rows* \times *B.columns* matrix
4. **for** *i* = 1 **to** *A.rows*
5. **for** *j* = 1 **to** *B.columns*
6. $c_{ij} = 0$
7. **for** *k* = 1 **to** *A.columns*
8. $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
9. **return** *C*

Identity Matrix

Identity function returns its n inputs as its outputs, corresponding matrix is the **identity matrix**

$$I_n = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

The Inverse

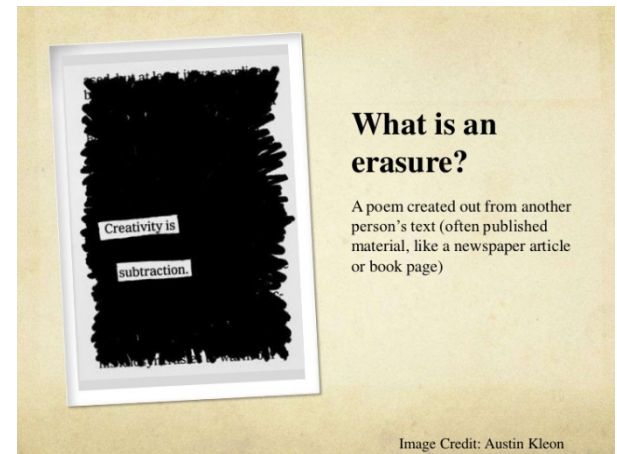
M a square matrix ($n \times n$) and
 M^{-1} its inverse, if it exists

$$M * M^{-1} = M^{-1} * M = I_n$$

M Invertible

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} -2 & 1 \\ \frac{3}{2} & \frac{-1}{2} \end{pmatrix}$$

$$M * M^{-1} = M^{-1} * M = I_n$$



ERASURE CODES

(3, 2)-CODE EXAMPLE CONTINUED

Erasure Codes

$n = 3, m = 2$ from above example (3, 2)-code

$$\begin{aligned} p_0 &= d_0 + d_1 + d_2 \\ p_1 &= 1 * d_0 + 2 * d_1 + 3 * d_2 \end{aligned} \quad P = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix}$$

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data Reconstruction Matrix

A.k.a. encoding matrix

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear function that returns input data and parity bytes

Data Reconstruction Matrix

A.k.a. encoding matrix

The diagram illustrates the data reconstruction matrix equation. A blue rounded rectangle highlights the top-left portion of the matrix, which contains the identity matrix. A blue callout bubble points to this section with the text "Identity matrix".

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear function that returns input data and parity bytes

Data Reconstruction Matrix

A.k.a. encoding matrix

The diagram illustrates the data reconstruction matrix equation. It shows a vector of data and parity bytes on the left, followed by an equals sign, a 5x3 matrix in the center, and a vector of data bytes on the right, with a multiplication symbol between the matrix and the vector. Two blue callout bubbles point to specific parts of the matrix: one points to the top 3x3 submatrix, and the other points to the bottom 2x3 submatrix.

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Identity matrix

Parity matrix / encoding matrix fragment

Linear function that returns input data and parity bytes

Data Reconstruction Matrix

A.k.a. encoding matrix

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear function that returns input data and parity bytes

Data Loss Example

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data Loss Example

$$\begin{bmatrix} \text{X} \\ d_1 \\ \text{X} \\ a_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} \text{X} & \text{X} & \text{X} \\ 0 & 1 & 0 \\ \text{X} & \text{X} & \text{X} \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data Loss Example

$$\begin{bmatrix} \text{X} \\ d_0 \\ d_1 \\ \text{X} \\ a_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} \text{X} & \text{X} & \text{X} \\ 0 & 1 & 0 \\ \text{X} & \text{X} & \text{X} \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear function mapping data bytes to non-lost data and parity bytes:

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data Loss Example

$$\begin{bmatrix} \text{X} \\ d_0 \\ d_1 \\ \text{X} \\ a_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} \text{X} & \text{X} & \text{X} \\ 0 & 1 & 0 \\ \text{X} & \text{X} & \text{X} \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} * \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Linear function mapping data bytes to non-lost data and parity bytes:

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

On the wrong side of equation ☹

Solve for d_i s

Need Inverse of Reconstruction Matrix

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \quad M^{-1} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

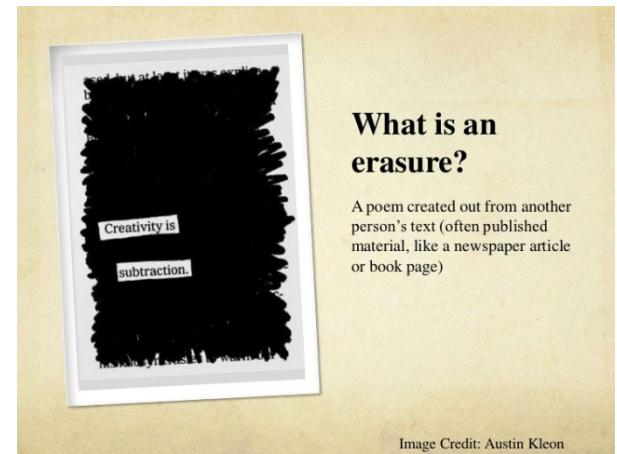
Solve for d_i s

Need Inverse of Reconstruction Matrix

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \quad M^{-1} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

Gives us **original data bytes** in terms of
known data and parity bytes!

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix} \begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix}$$



ERASURE CODES

GENERALIZING TO (N, M) -CODES

Generalizing

Arbitrary n, m - compute parity matrix

Generate an $m \times n$ **parity matrix P** (rows need to be “*sufficiently different*”)

$$\begin{bmatrix} p_0 \\ \vdots \\ p_{m-1} \end{bmatrix} = \begin{pmatrix} p_0 \\ \vdots \\ p_{m-1} \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

p_i are rows of P – an $m \times n$ parity matrix

Compute Data Reconstruction Matrix

Append rows of P to I_n , denoted as e_i

$$\begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \\ p_0 \\ \vdots \\ p_{m-1} \end{bmatrix} = \begin{pmatrix} e_0 \\ \vdots \\ e_{n-1} \\ p_0 \\ \vdots \\ p_{m-1} \end{pmatrix} * \begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

Data Loss with Resulting Matrix

- Indices $k \leq m$ of **missing data bytes** are i_0, \dots, i_{k-1}
- Remove rows corresponding to missing data bytes
- Keep k **parity rows** p_0, \dots, p_{k-1}
- j_0, \dots, j_{n-k-1} indices of present $n-k$ data bytes

$$\begin{bmatrix} d_{j_0} \\ \vdots \\ d_{j_{n-k-1}} \\ p_0 \\ \vdots \\ p_{k-1} \end{bmatrix} = \begin{pmatrix} e_{j_0} \\ \vdots \\ e_{j_{n-k-1}} \\ p_0 \\ \vdots \\ p_{k-1} \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

Compute M 's Inverse M^{-1}

Reconstruct data by multiplying with M^{-1}

If P was chosen correctly, M^{-1} exists.

$$\begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \end{bmatrix} = M^{-1} * \begin{bmatrix} d_{j0} \\ \vdots \\ d_{jn-k-1} \\ p_0 \\ \vdots \\ p_{k-1} \end{bmatrix}$$

Loose Ends

- *How do we compute matrix inverses?*
- *How do we generate “optimal” parity matrices P s.t. M^{-1} always exists?*
- *How do we compute parity bytes instead of parity numbers (informally referred to as bytes, above)?*

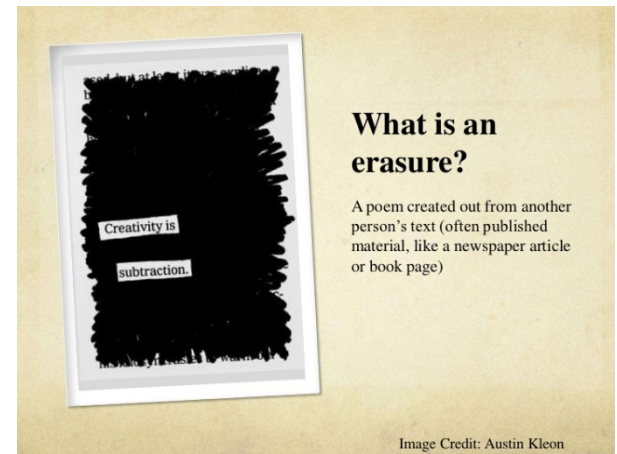
Facts: Non-invertible Matrix

More a negative result

- M has a row that can be expressed as a linear combination of other rows of M then M is non-invertible
- Said differently:
 - Linear function corresponding to M has one of its outputs redundant with the other outputs, so it is essentially a linear function taking n inputs to fewer than m outputs

“Not Sufficiently Different”

- If one parity function is a linear combination of other parity functions then it is redundant, i.e., not sufficiently different
- Therefore, choose P wisely
- **No row of P should be expressed as linear combination of other rows of P**
- ...



ERASURE CODES

ANOTHER EXAMPLE

Erasure Coding Example

- Storage: “My private key”
- Pad with spaces to obtain right length if needed
 - “My private key__” (added two spaces to obtain length of 16 characters)
- Build data matrix, here, a 4x4 matrix (ASCII code)

```
My p
riva
te k
ey__
```

Encoding Matrix

Protect against loss of 2 bytes

- Identity matrix appended with
- Parity matrix
 - Derived from parity equations, i.e., one per parity byte (a.k.a. encoding matrix fragment)
- Results in full encoding matrix

Encoding Matrix

(For a (3, 2)-code)

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Parity equations need to be “sufficiently different,” e.g., not be linear combinations of each other.

Encoding Matrix

(For a (3, 2)-code)

The diagram shows the encoding equation for a (3, 2)-code. On the left, a column vector of five elements is shown: $\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix}$. This is followed by an equals sign and a 5x3 matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix}$. To the right of the matrix is a dot and another column vector: $\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$. A blue rounded rectangle highlights the top three rows of the matrix (rows 1, 2, and 3). A blue callout bubble points to this highlighted section with the text "Identity matrix".

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Identity matrix

Parity equations need to be “sufficiently different,” e.g., not be linear combinations of each other.

Encoding Matrix

(For a (3, 2)-code)

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$
$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Identity matrix

Encoding matrix fragment – derived from parity equations

Parity equations need to be “sufficiently different,” e.g., not be linear combinations of each other.

Parity Matrix

- Results from **multiplying encoding matrix with original data matrix**

A diagram illustrating the multiplication of two matrices. On the left, a large blue square bracket represents a matrix with dimensions $p \times q$, labeled "# columns" and $p \times q$. A horizontal blue rectangle is drawn inside this bracket. To its right is a small blue asterisk. Further right is a tall, narrow blue square bracket representing a matrix with dimensions $q \times r$, labeled "# rows" and $q \times r$. To its right is a small blue equals sign. Further right is another tall, narrow blue square bracket representing the result matrix with dimensions $p \times r$, labeled "# result" and $p \times r$. A small blue square is drawn inside this final bracket.

$$\begin{bmatrix} \text{# columns} \\ p \times q \end{bmatrix} * \begin{bmatrix} \text{# rows} \\ q \times r \end{bmatrix} = \begin{bmatrix} \text{# result} \\ p \times r \end{bmatrix}$$

Data Loss Example

$$\begin{bmatrix} \overset{\times}{d_0} \\ d_1 \\ \overset{\times}{d_2} \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} \overset{\times}{1} & \overset{\times}{0} & \overset{\times}{0} \\ 0 & 1 & 0 \\ \overset{\times}{0} & \overset{\times}{0} & \overset{\times}{1} \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data reconstruction matrix:

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data Loss Example

$$\begin{bmatrix} \cancel{d_0} \\ d_1 \\ \cancel{d_2} \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} \cancel{1} & \cancel{0} & \cancel{0} \\ 0 & 1 & 0 \\ \cancel{0} & \cancel{0} & \cancel{1} \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Data reconstruction matrix:

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix}$$

Recovering Lost Data

- Find **Inverse** to data reconstruction matrix (from previous slide; it is a **square matrix**)
- For large matrix, use online tools to determine matrix inverse
- Recover data by multiplying **Inverse with relevant rows of parity matrix** (rows not affected by data loss)

Solve for d_i s

Need Inverse of Data Reconstruction Matrix

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \quad M^{-1} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix} \begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix}$$

Solve for d_i s

Need Inverse of Data Reconstruction Matrix

$$\begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \quad M^{-1} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

Gives us original data bytes in terms of known data and parity bytes!

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{pmatrix} \frac{-1}{2} & \frac{3}{2} & \frac{-1}{2} \\ 1 & 0 & 0 \\ \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \end{pmatrix} \begin{bmatrix} d_1 \\ p_0 \\ p_1 \end{bmatrix}$$