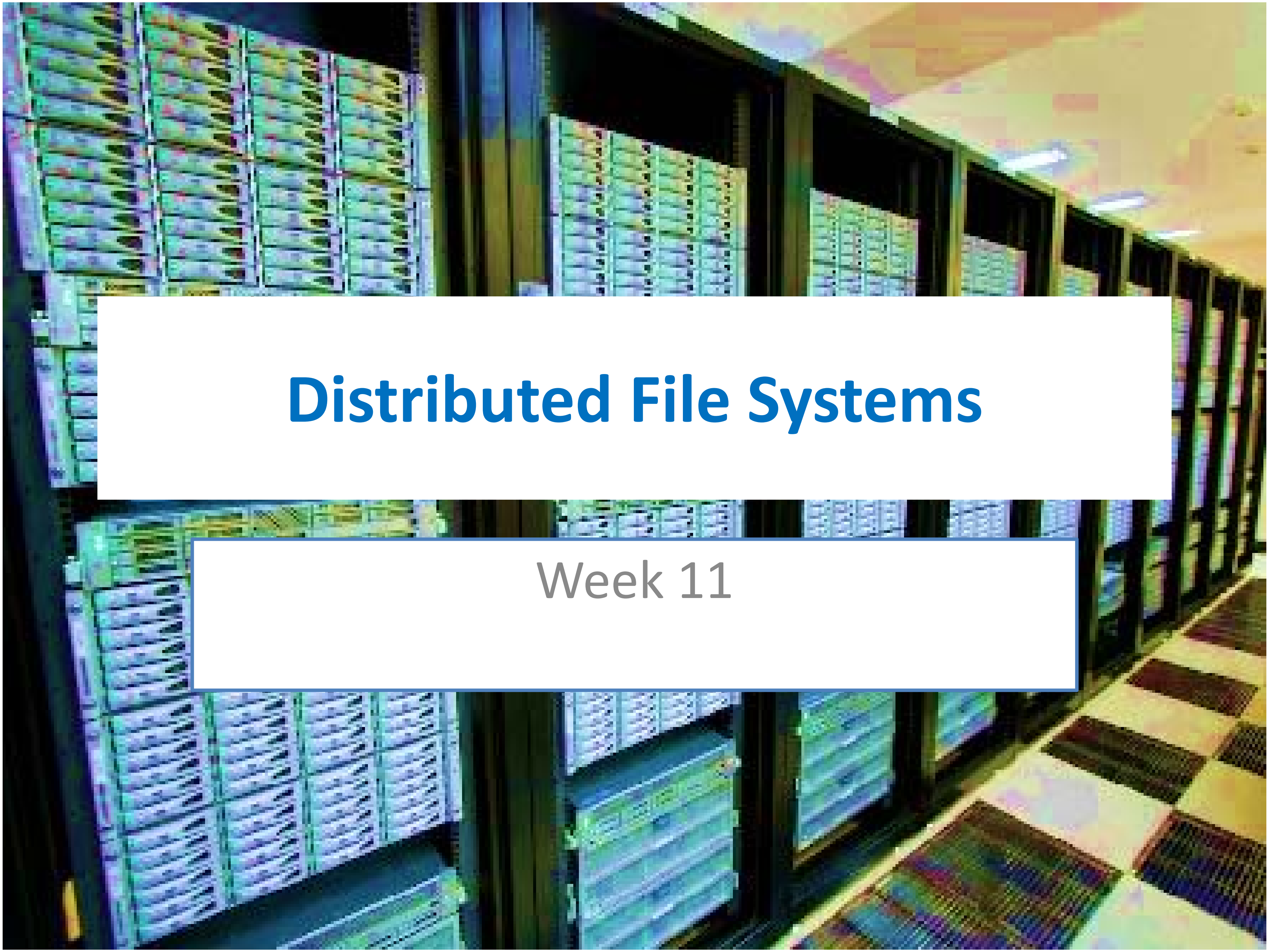
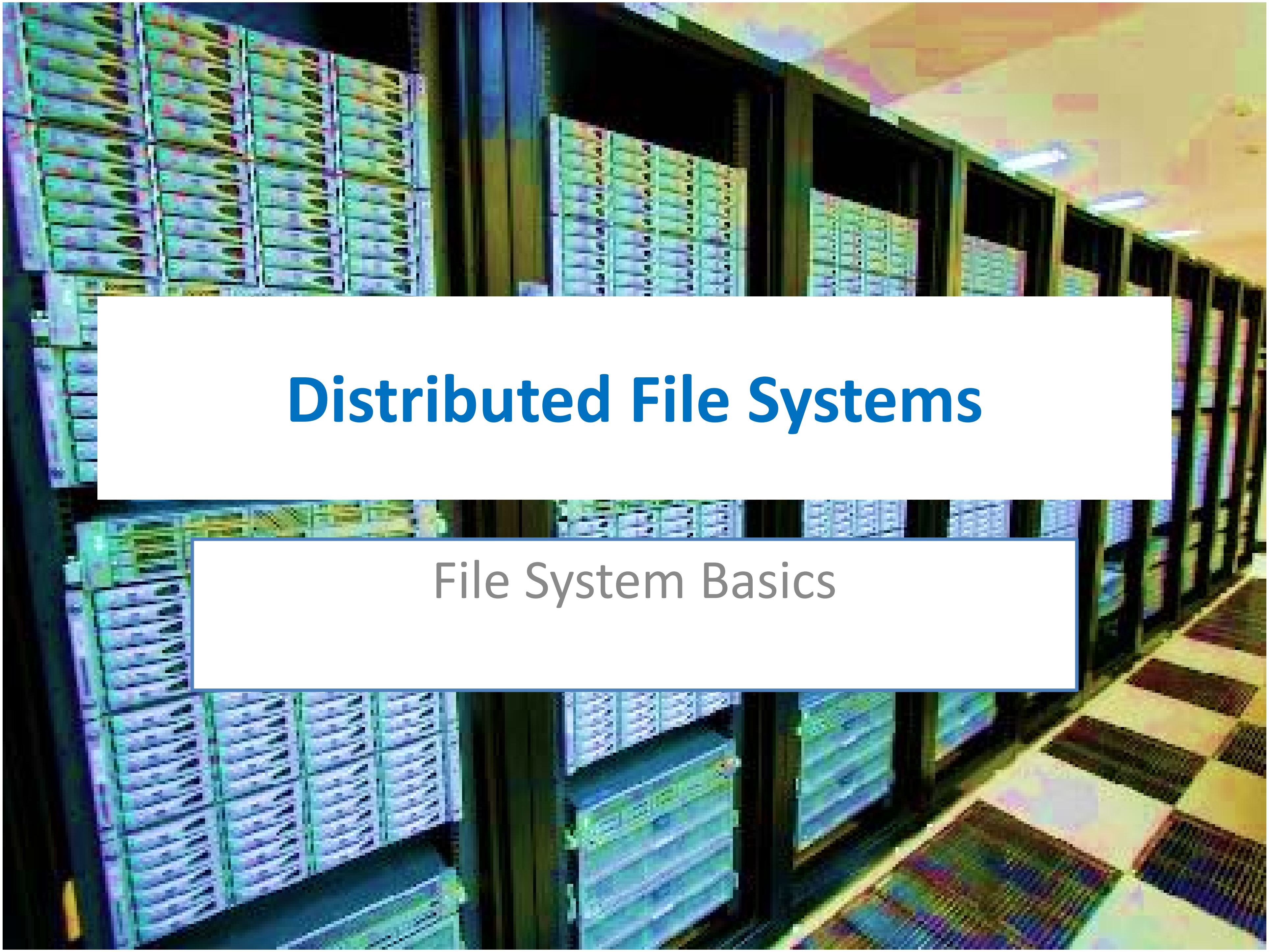


Distributed File Systems

Week 11

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

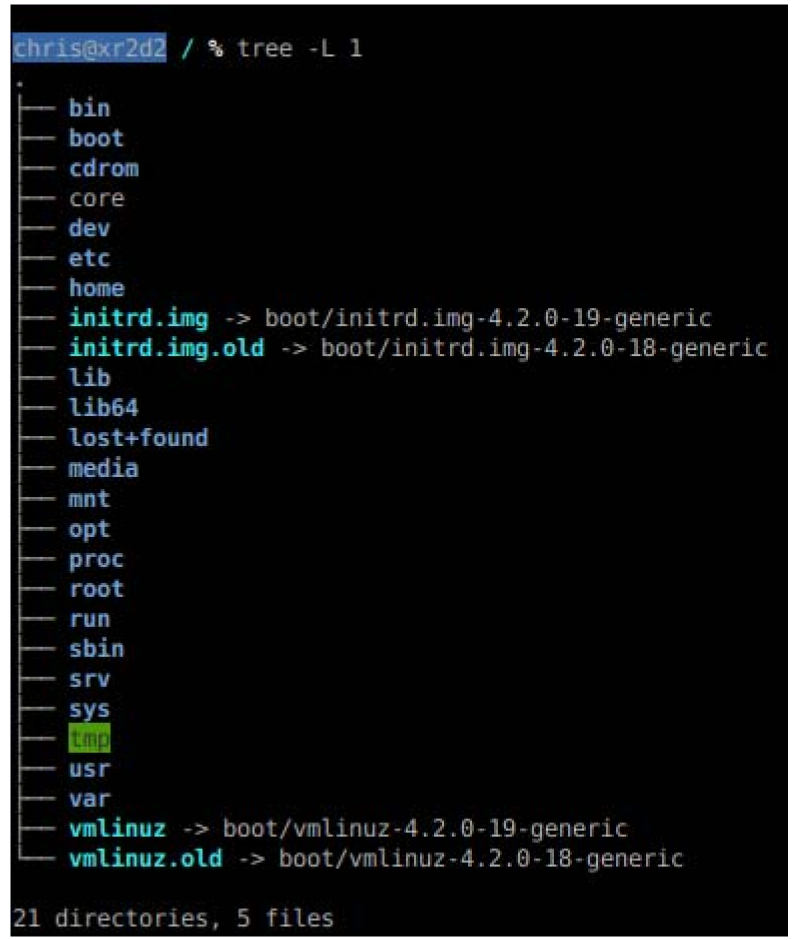
Interaction with file systems

- 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



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

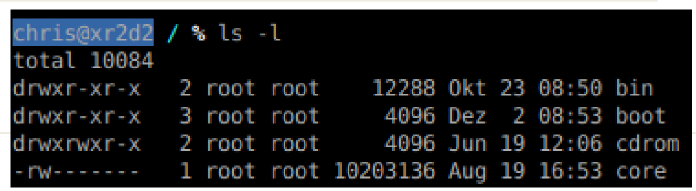
```
Types of locks:
F_RDLCK  Shared or read lock
F_WRLCK  Exclusive or write lock
F_UNLCK  Unlock
```

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**

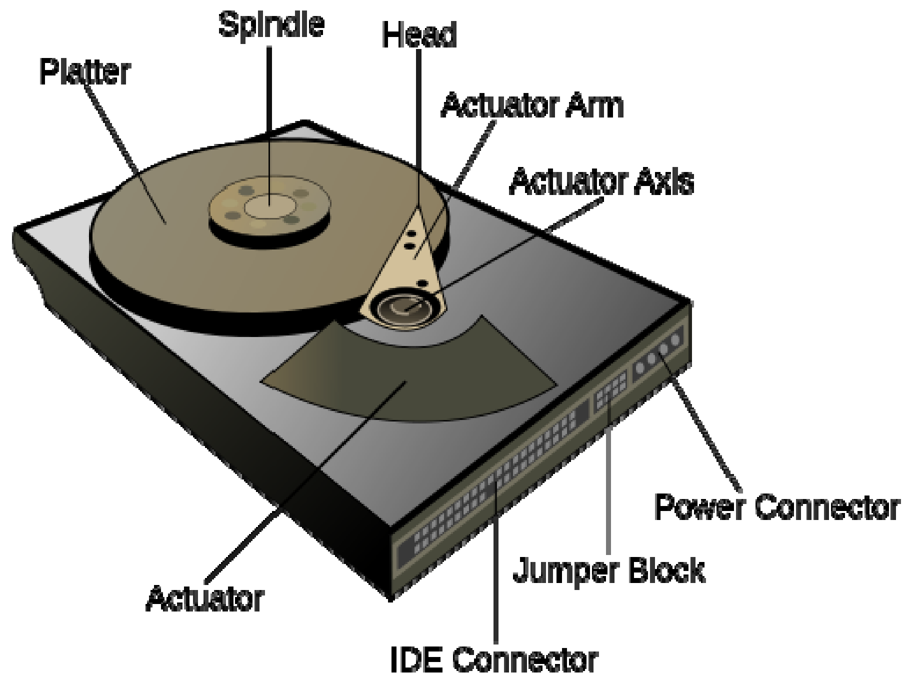
```
// set user read and write permission for file.txt
chmod(“file.txt”, S_IRUSR | S_IWUSR)
```




```
//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



Source:Wikipedia

- 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)

Source:OCZ

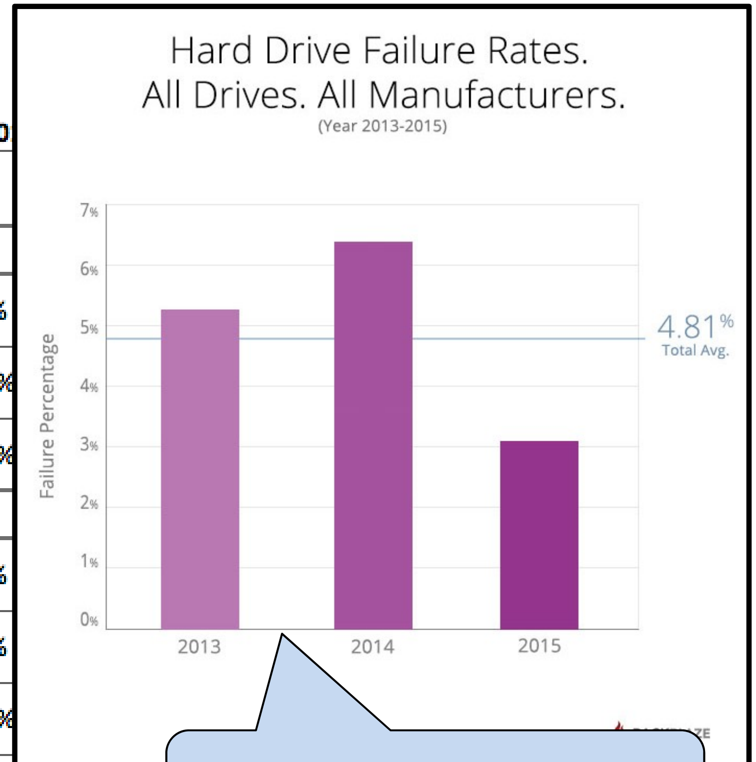
- Can also be connected via PCI Express
- Low-level operations differ a lot compared to HDD
  - On SSD's overwriting costs more → TRIM Command
  - Deleting is delegated to internal firmware which has a garbage collector

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	All Periods: 20 Failure Rate	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%	293	
Toshiba DT01ACA Series (TOSHIBA DT01ACA300)	3TB	6.93%	3.68%	2.80%	4.20%	1.40%	9.80%	58	[illegible]
Western Digital Red 3 TB (WDC WD30EFRX)	3TB	3.79%	6.94%	8.79%	7.65%	6.40%	9.30%	1,085	16.3
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%**				



~5% fail per year

~10% failed during first 3 years

Source: https://www.backblaze.com/blog/hard-drive-reliability-q3-2015/

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

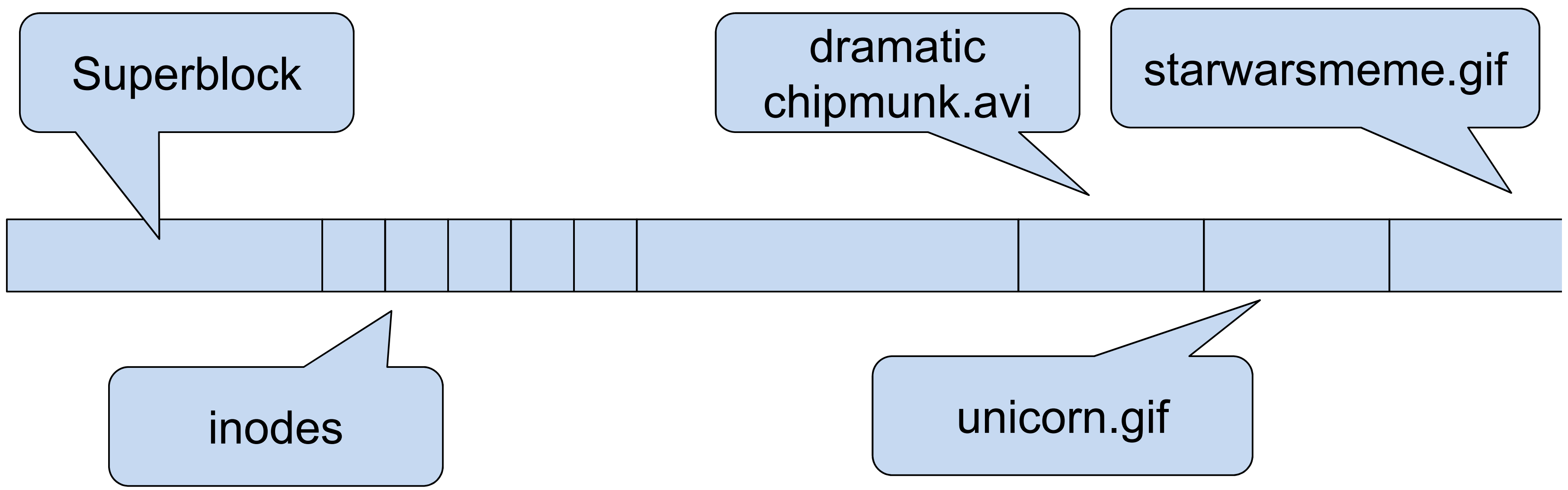
Disk file systems

- 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**



ext2 inode

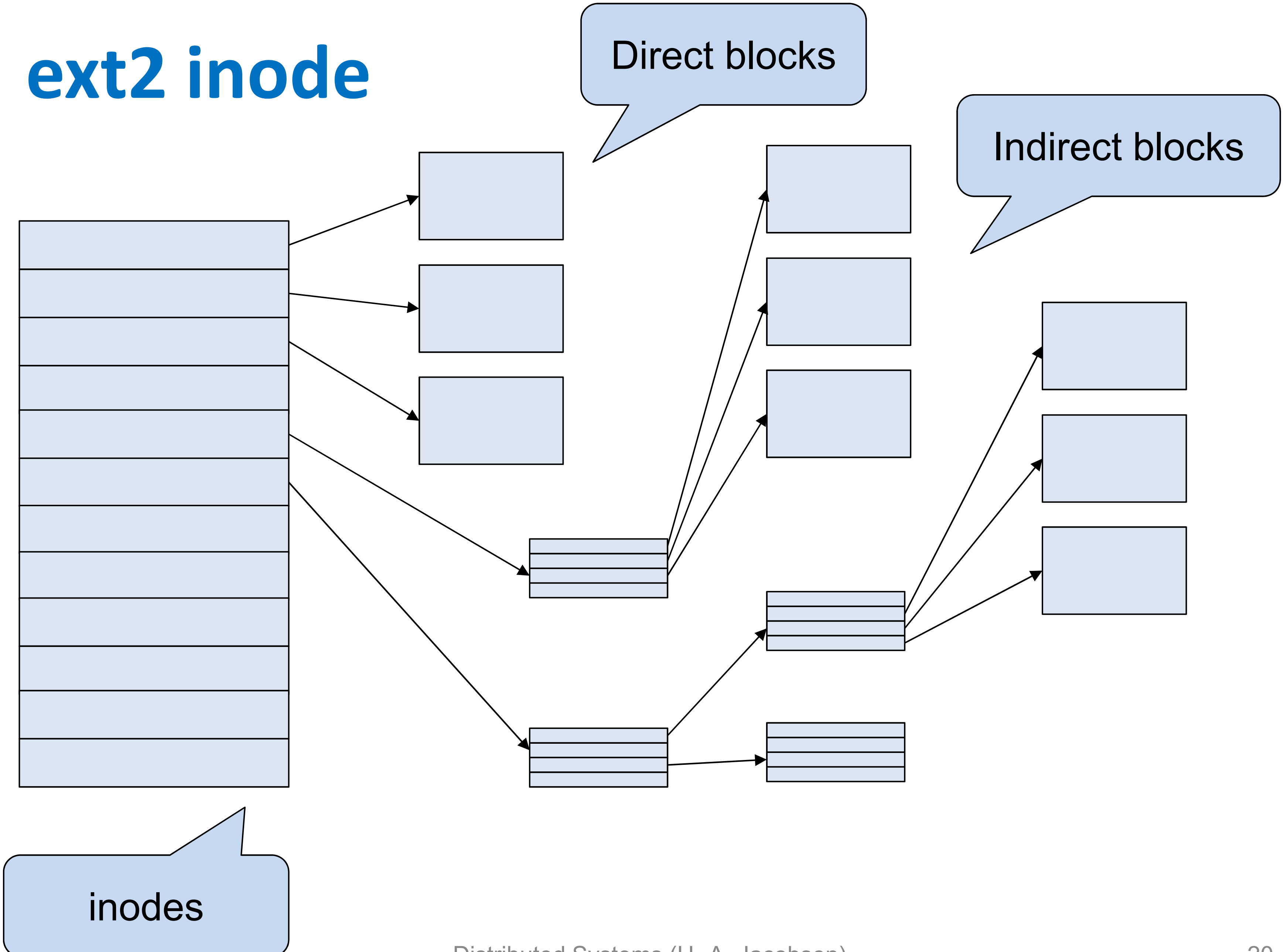
- Owner and group identifiers
- File length
- File type and access rights
- Number of data blocks
- Array of pointers to data blocks
- Timestamp

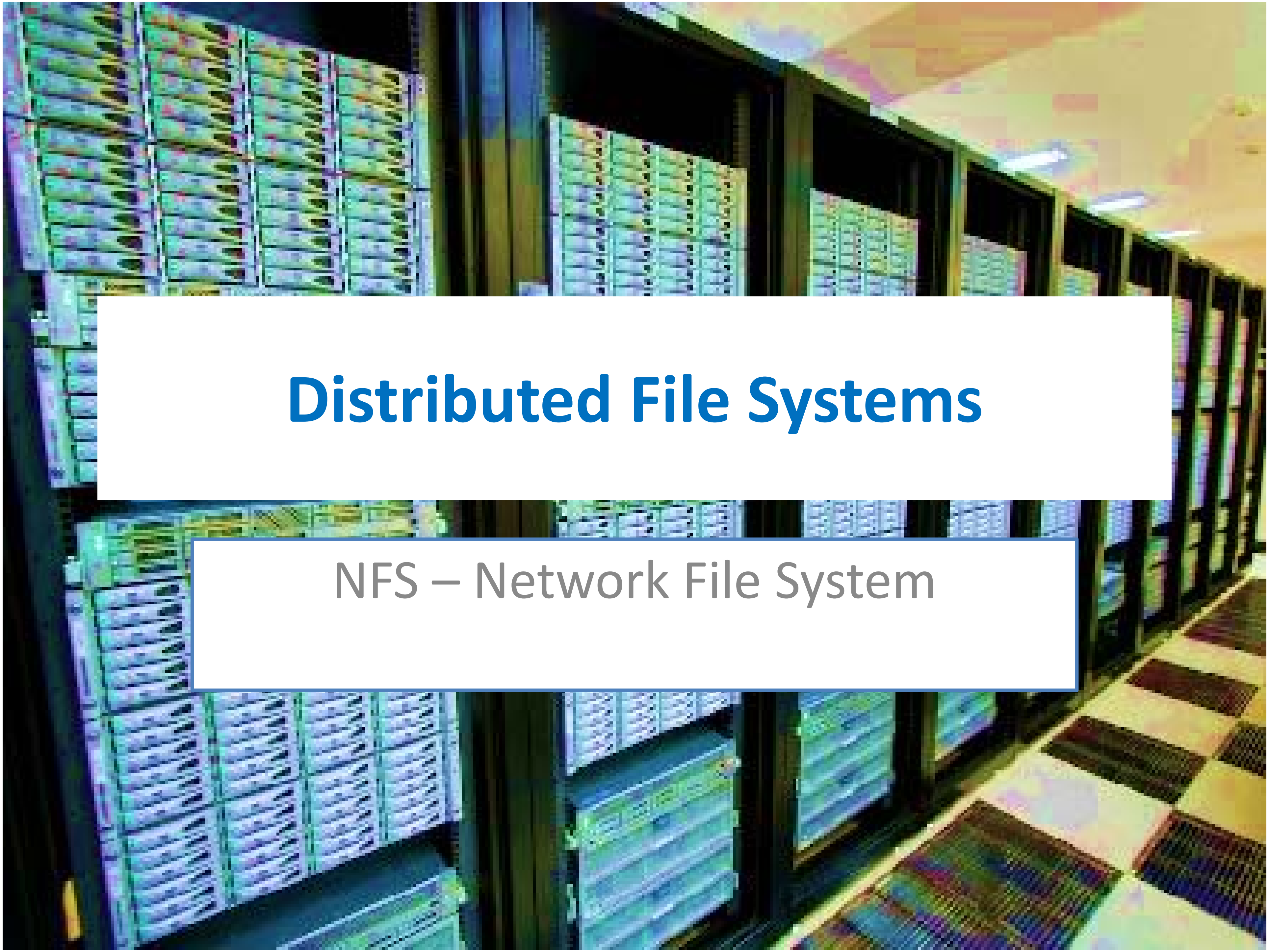
- Types
  - File
  - Directory
  - Symbolic link

```
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





Distributed File Systems

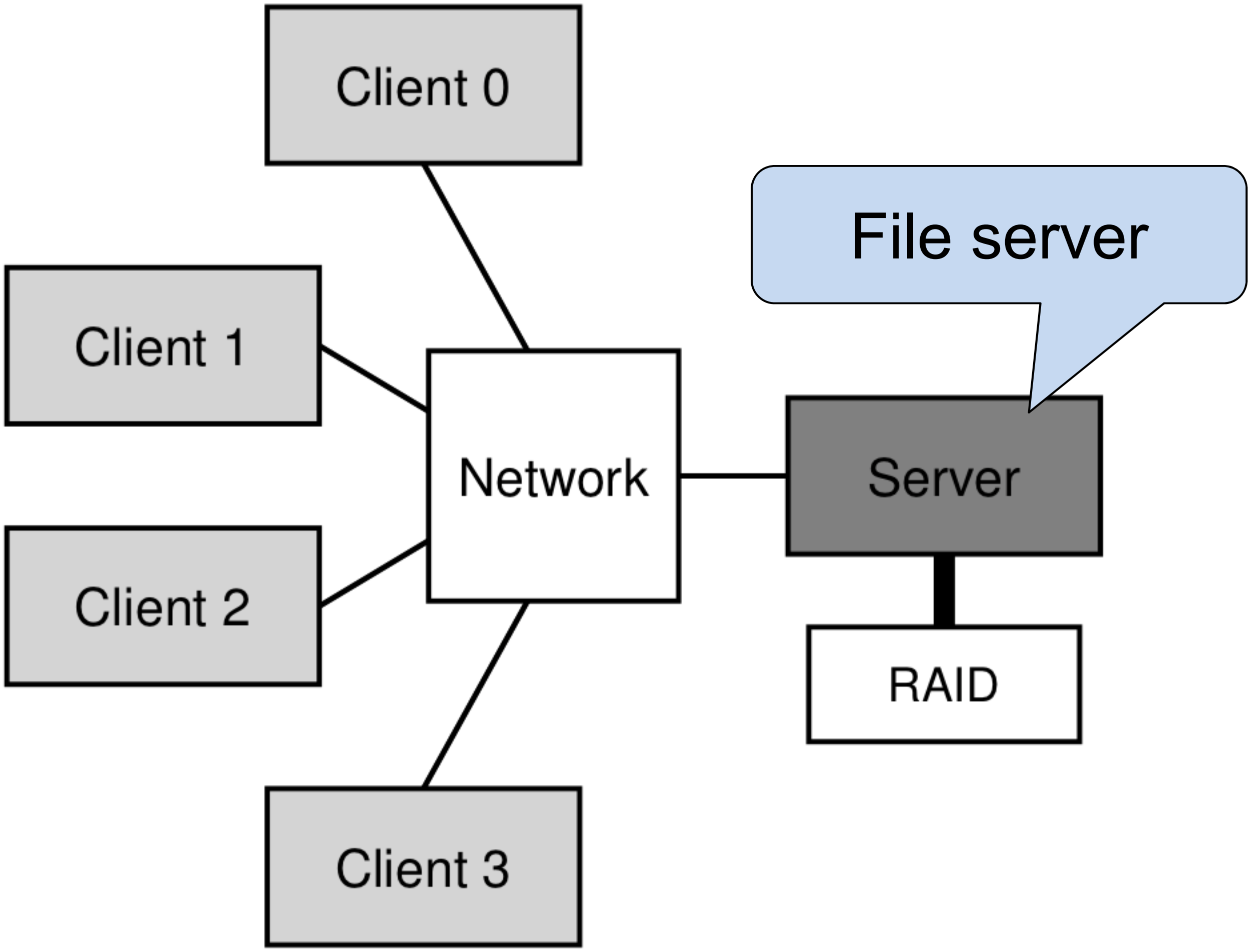
NFS – Network File System

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



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 accesed by **clients**
  - Namespace is **distributed** across servers
  - Each client treats remote files as local ones (“virtual files“)

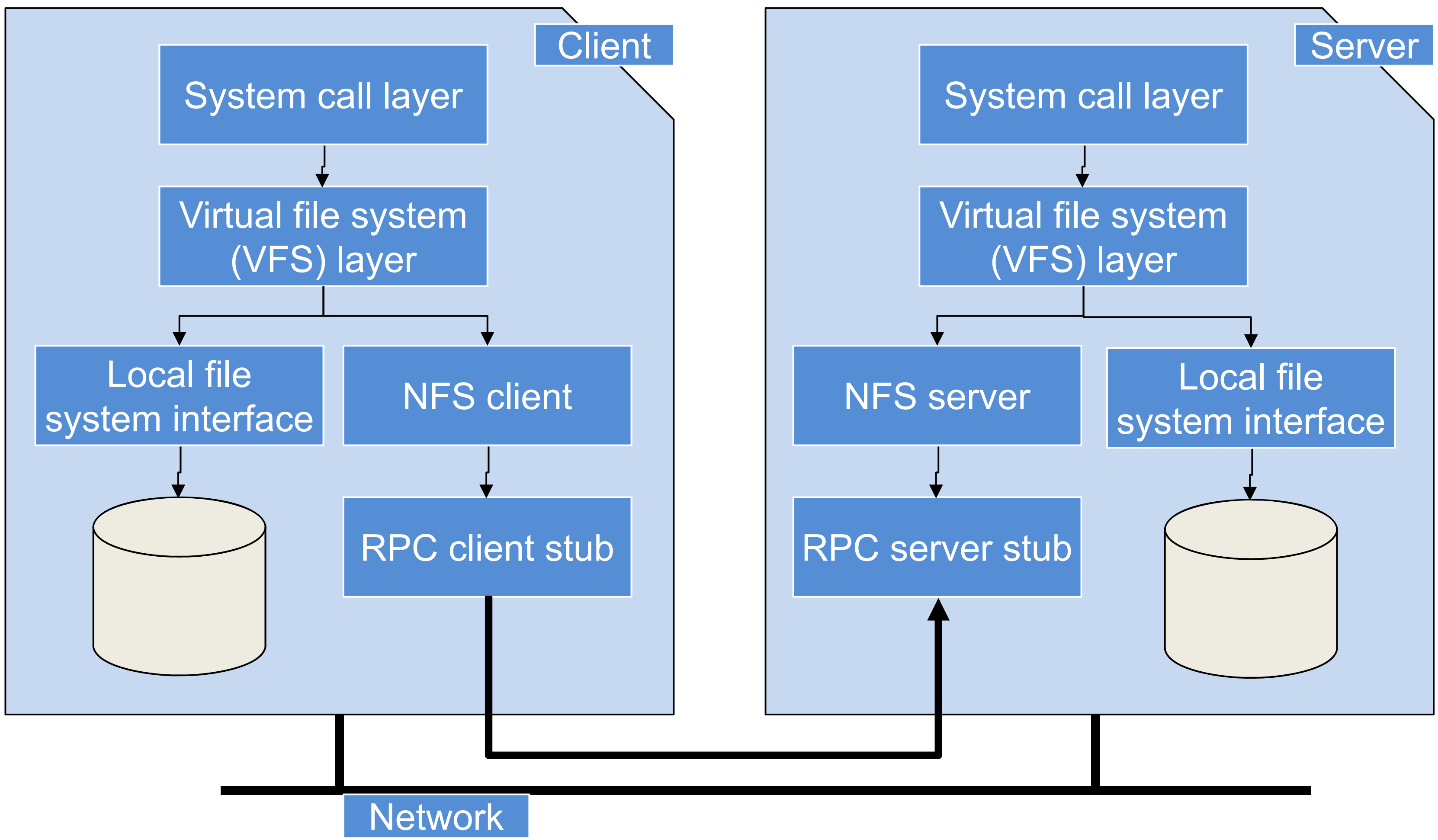
The Network File System (NFS)

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

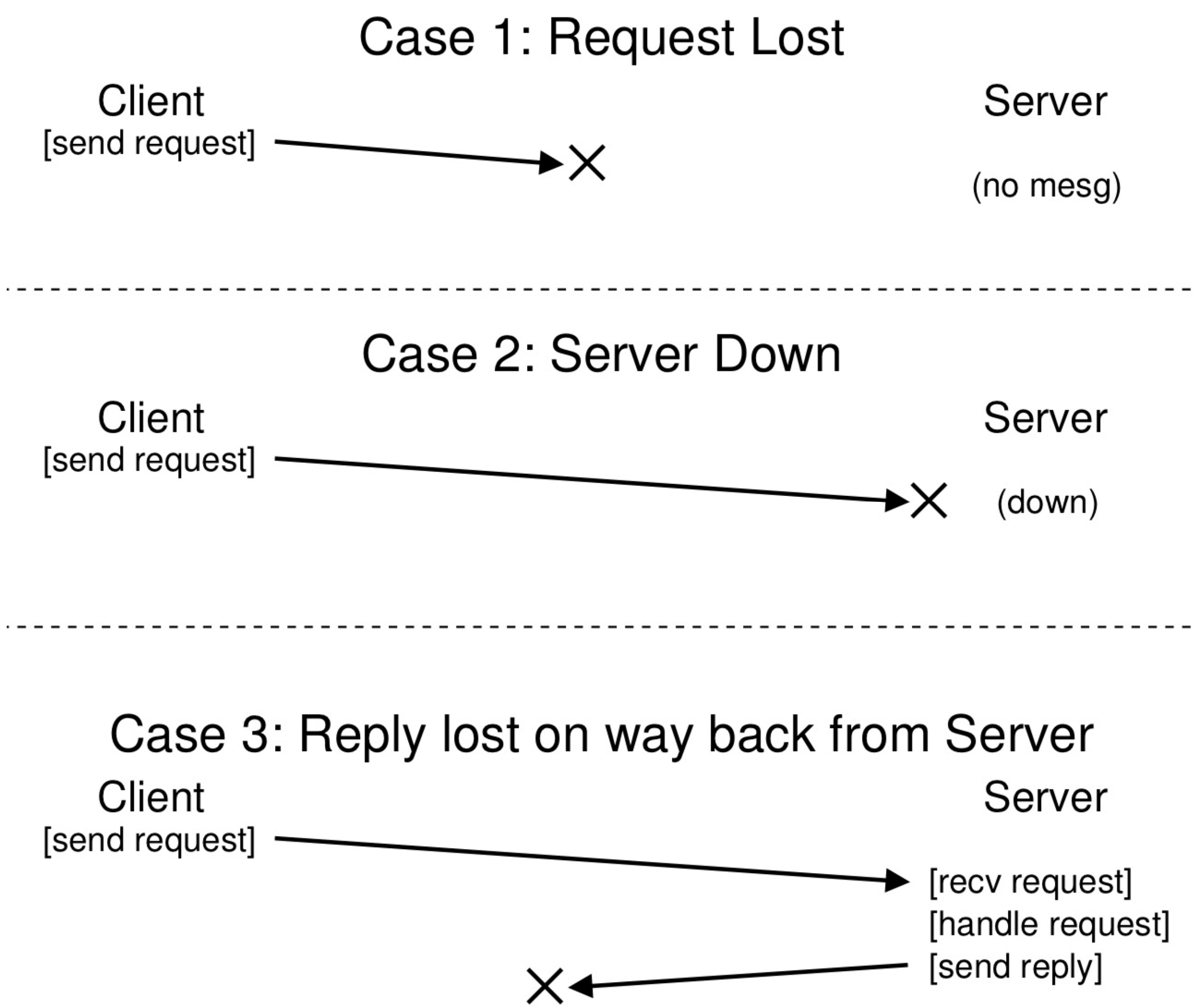
- RPC commands are **stateless**: server does not maintain state across commands in a "session"
- *read()* is stateful (server needs to remember *seek()*) → **read(position) is stateless** (server has all the information needed for correct read)
- With stateless RPC, server can fail and later continue 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



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

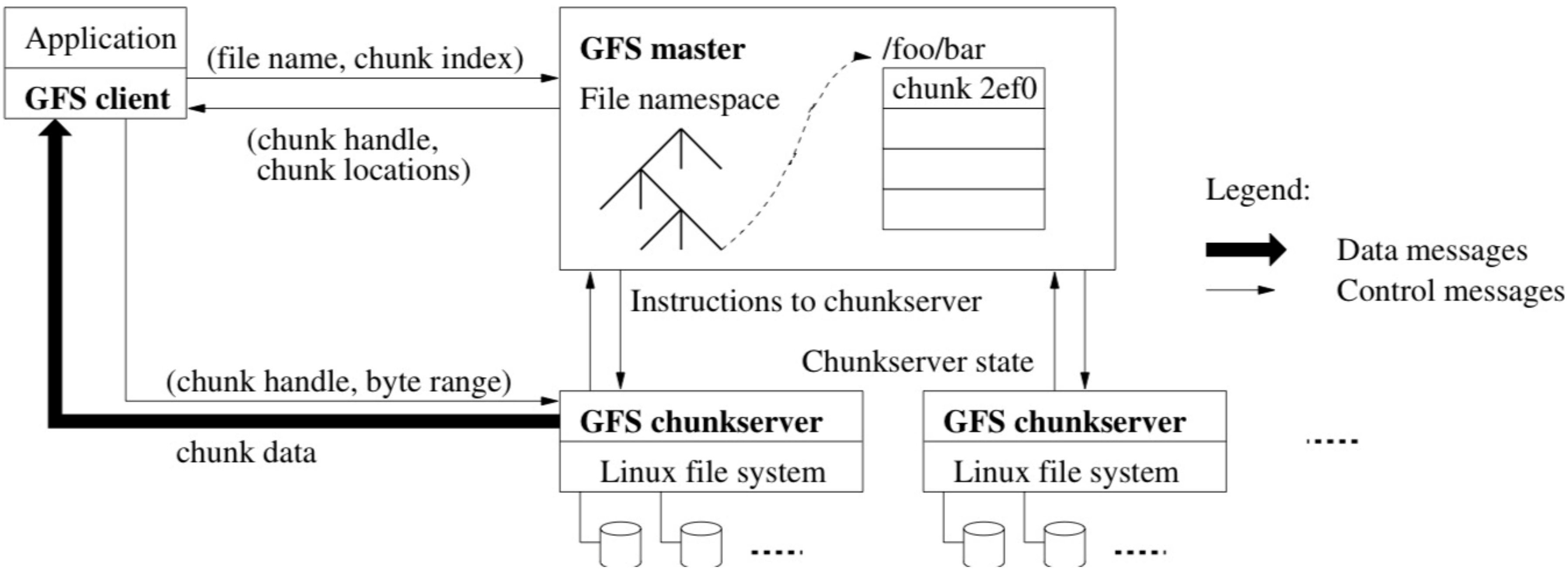
Interface

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

Architecture

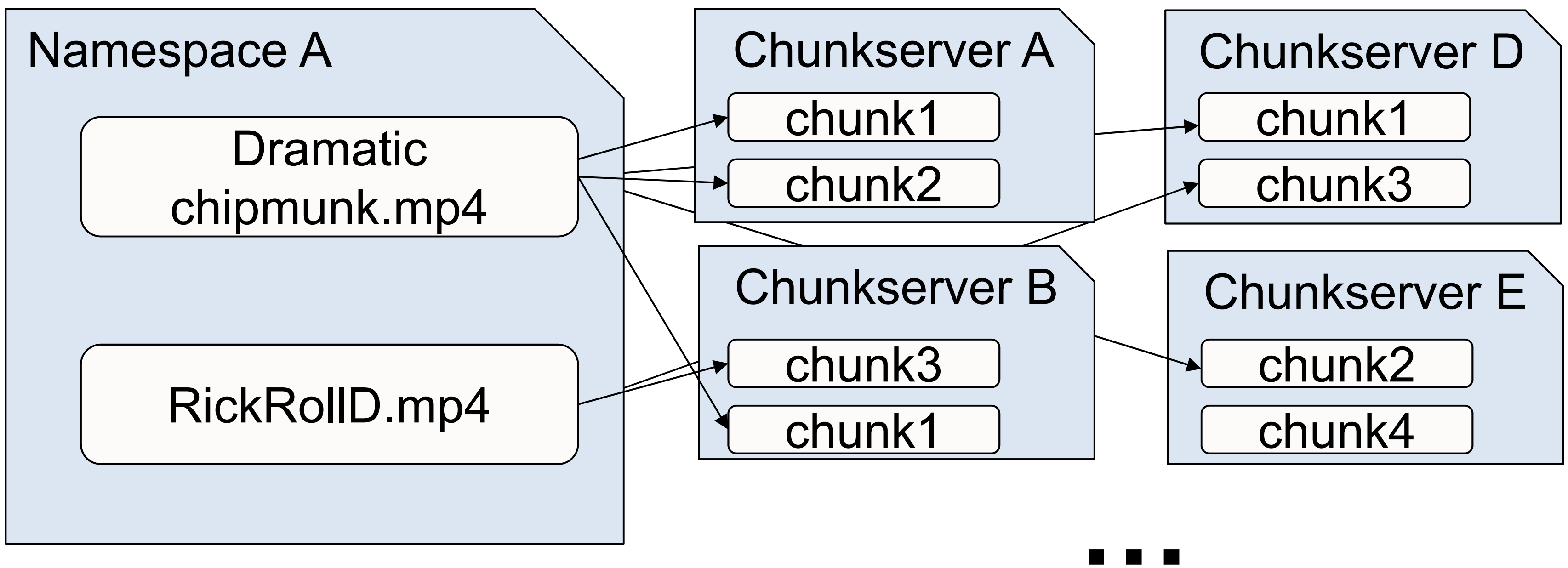
- Files
  - Divided into fixed-size chunks (64MB)
  - Identified by an immutable and unique id (chunk handle)
- Single master
  - Maintains file system metadata
    - Namespace, access control information, mapping from files to chunks, location of chunks
  - Heartbeats chunkservers
- Multiple chunkservers
  - 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 log to minimize replaying effort

How is fault-tolerance achieved?

- Master
  - Operation log, replication 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
- Heartbeat messages
  - Chunkservers can fail any time
  - Can trigger re-replication
  - Share current load
  - Can 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