Chapter 12:
Distributed File Systems

Distributed File Systems

- Objectives
  - To be reminded on storage systems and their properties
  - To understand the basics of file system modules and attributes
  - To compare differences of file systems and their distributed sisters as well as understand major characteristics of examples
- Topics
  - Storage, file systems, and file attribute records
  - Reference counting
  - Distributed file systems: Operations and requirements
  - File service architecture
    - Network File System (NFS)
    - Inter Planetary File System (IPFS)
    - Google File System

Storage Systems and Their Properties

System	*Sharing across network*	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	2 ✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	2	Ivy
Remote objects (RMI)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	3	OceanStore

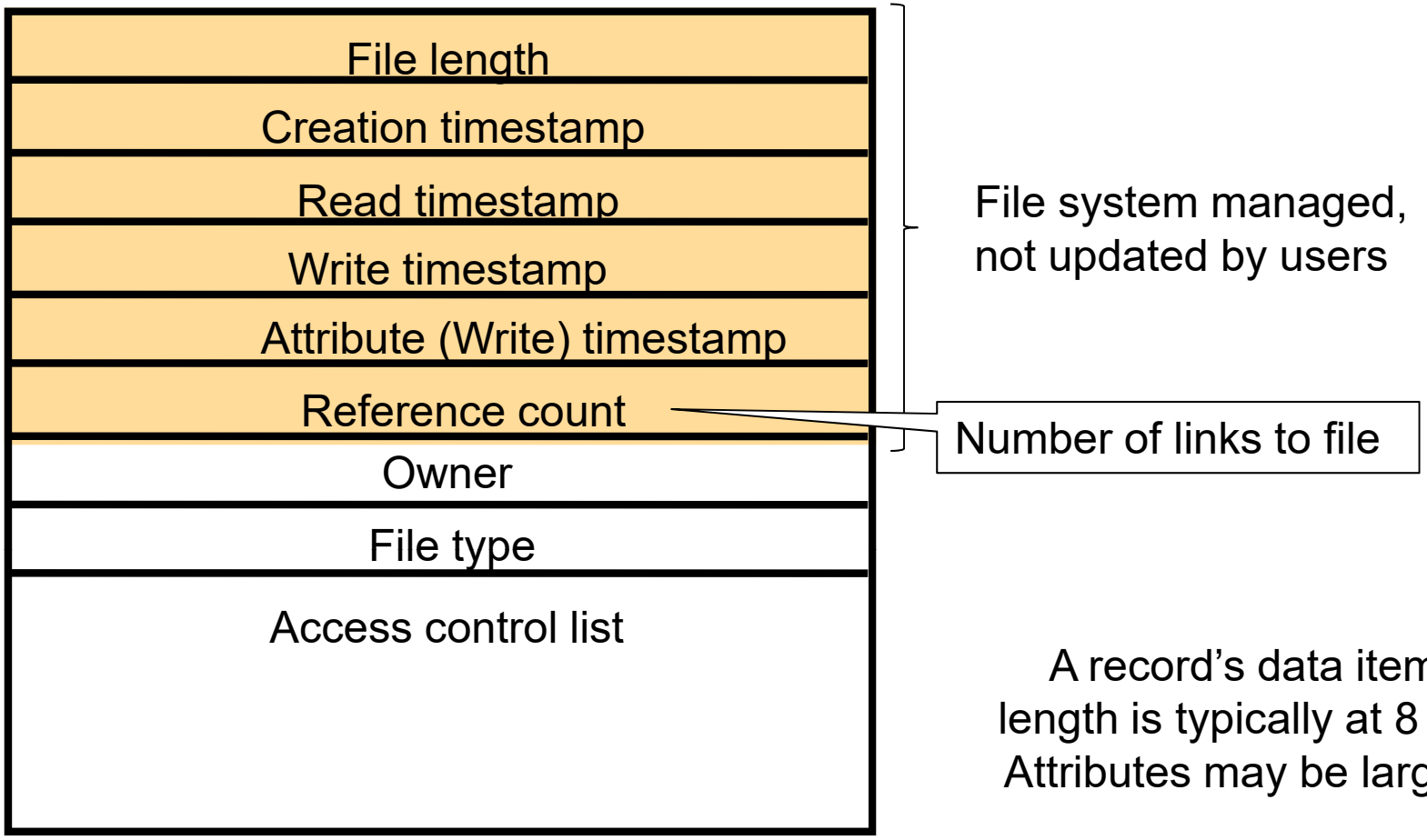
Consistency types: 1: Strictly one-copy
✓ 2: Slightly weaker guarantees
3: Considerably weaker guarantees

File System Modules

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

File Attribute Record Structure

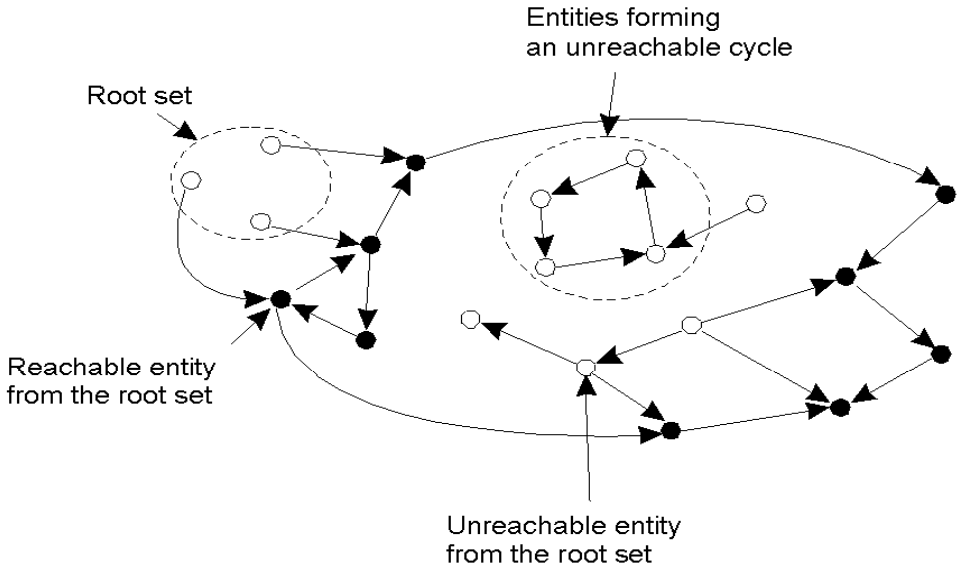
File systems: organization, storage, retrieval, naming, sharing, protection of files
Files: contain attributes (maintained by file system) and data (updated by user)



A record's data item length is typically at 8 bit. Attributes may be larger.

Reference Counting

- Maintenance task for distributed systems, OSes
  - Applied to files and objects
- *E.g.*, garbage collection
  - Delete object/file, if not reachable from root

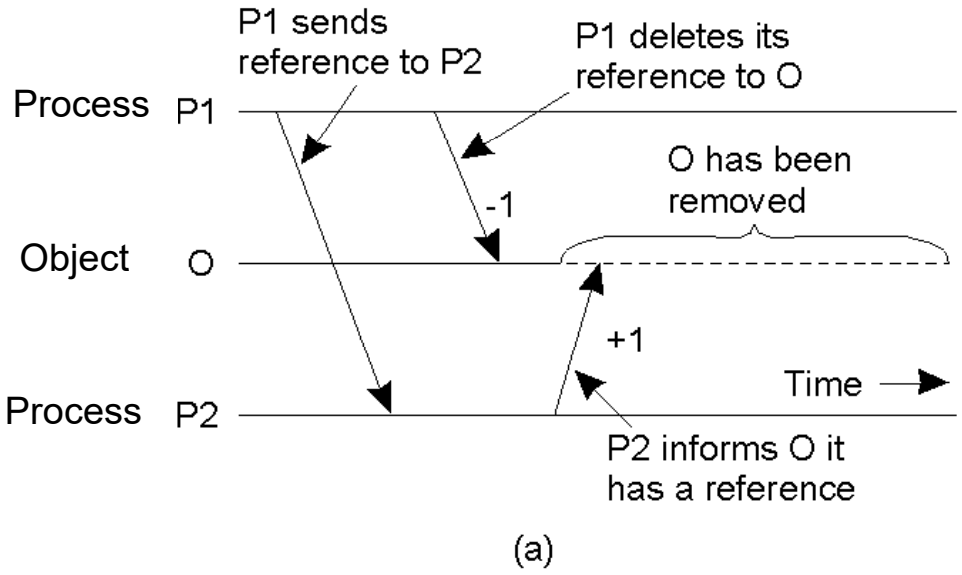


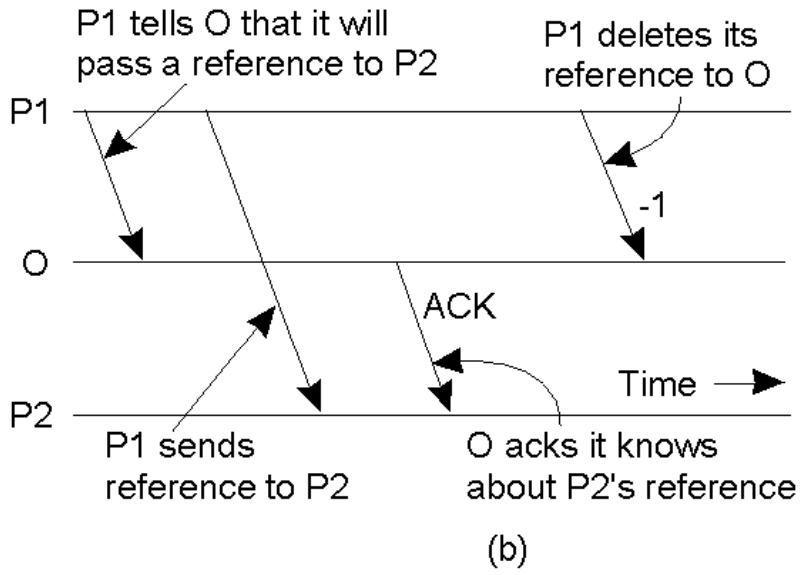
- How to run such a scheme in a distributed manner?

Distributed Reference Counting

- Copying a reference to another process and incrementing the counter can be too late

- A solution





UNIX File System Operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Figure 12.4, CDK

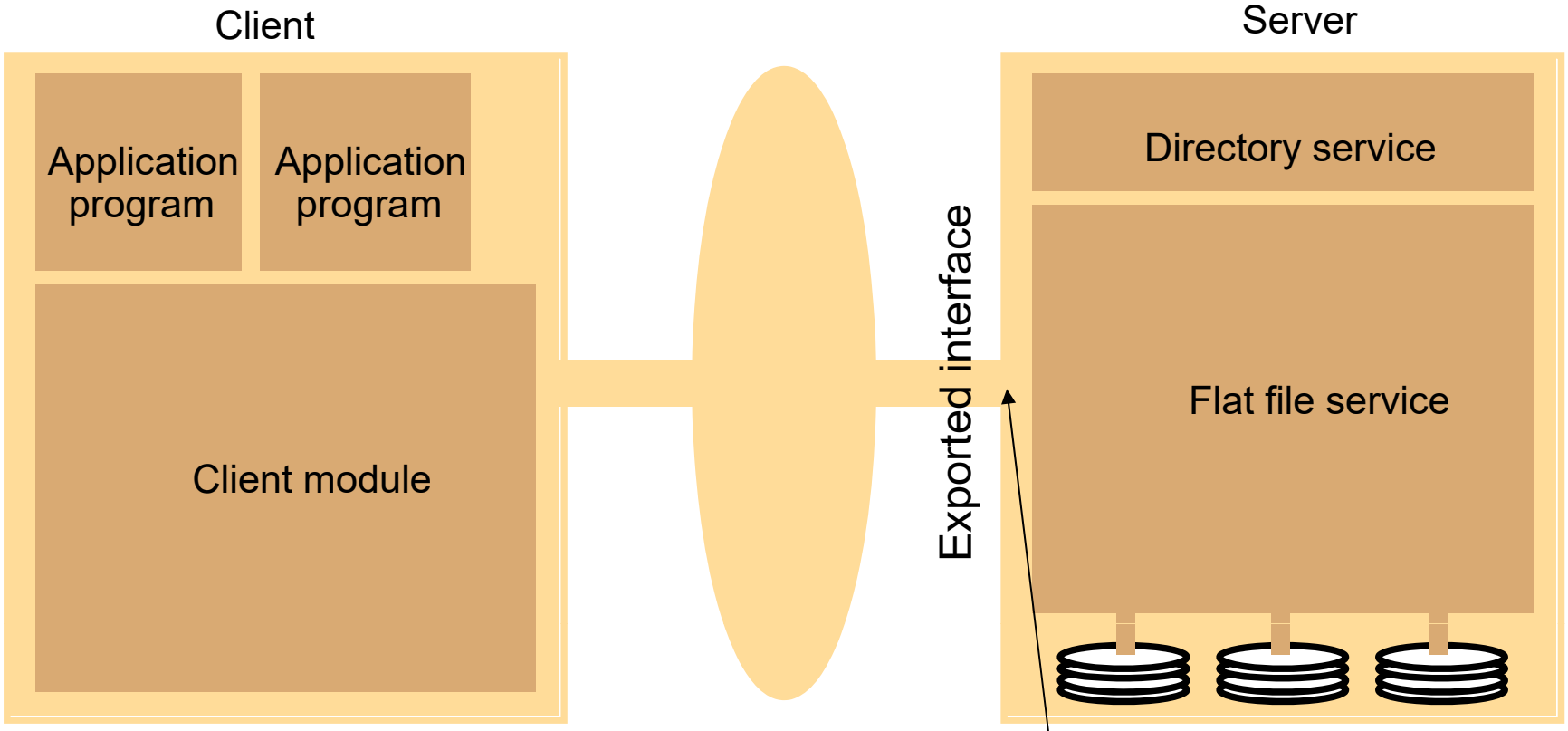
Distributed File System Requirements

- Transparency
  - Access, Location, Scaling, and all the others

- Concurrent file updates
- File replication
- Hardware and software heterogeneity
- Fault tolerance
- Consistency (one-copy update semantics)
- Security
- Efficiency

File Service Architecture

- Services: read, write, create, delete, …



- Unique File Identifiers (UFID) used
  - Referring to files in all requests for service operations

Flat File Service Operations

Read(FileId, i, n) -> Data — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 8.3).

Note: This interface is repeatable (*i.e.*, idempotent) and stateless!

Figure 12.6, CDK

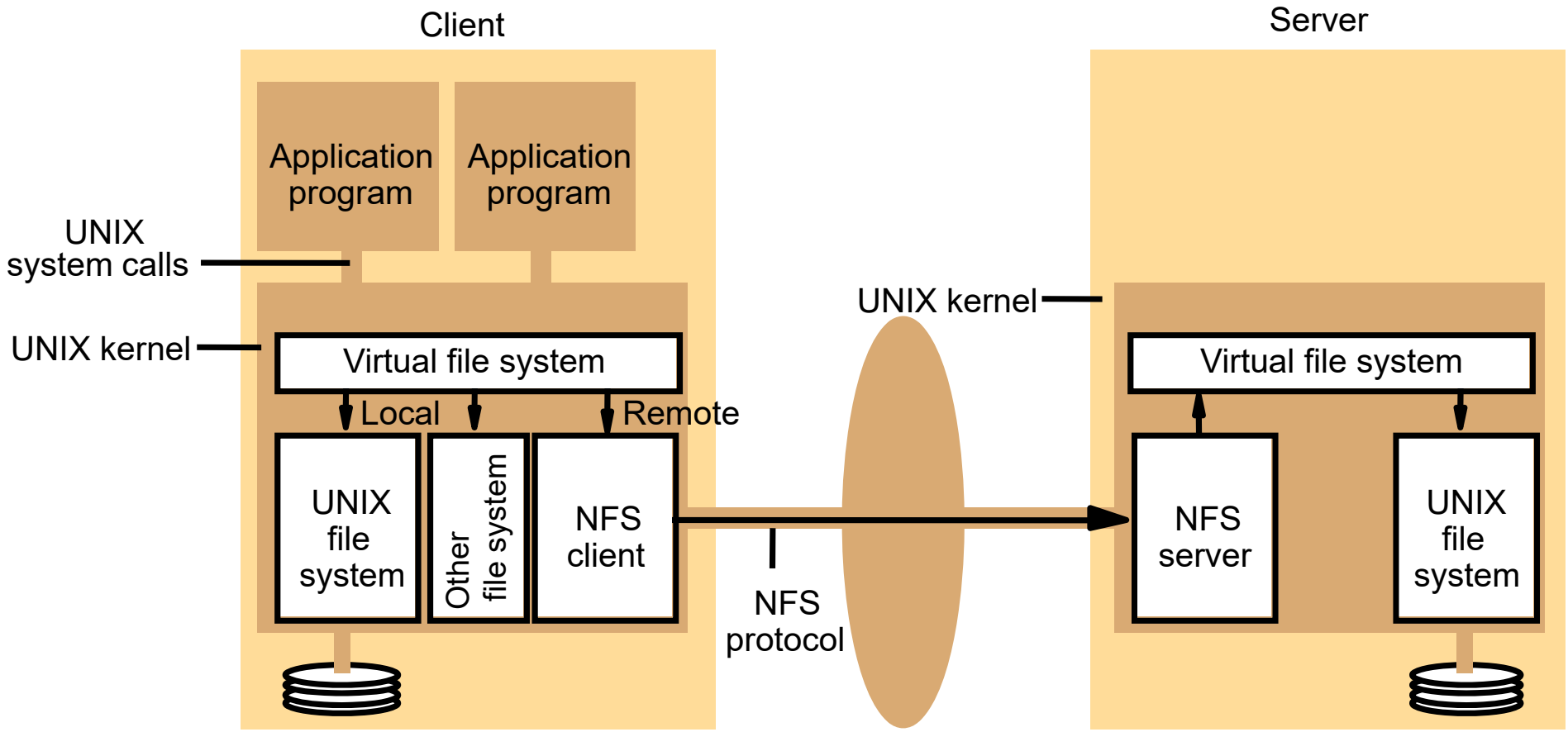
Directory Service Operations

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

Figure 12.7, CDK

Case Study 1: NFS Architecture

- Network File System (NFS)



NFS Server Operations Simplified (1)

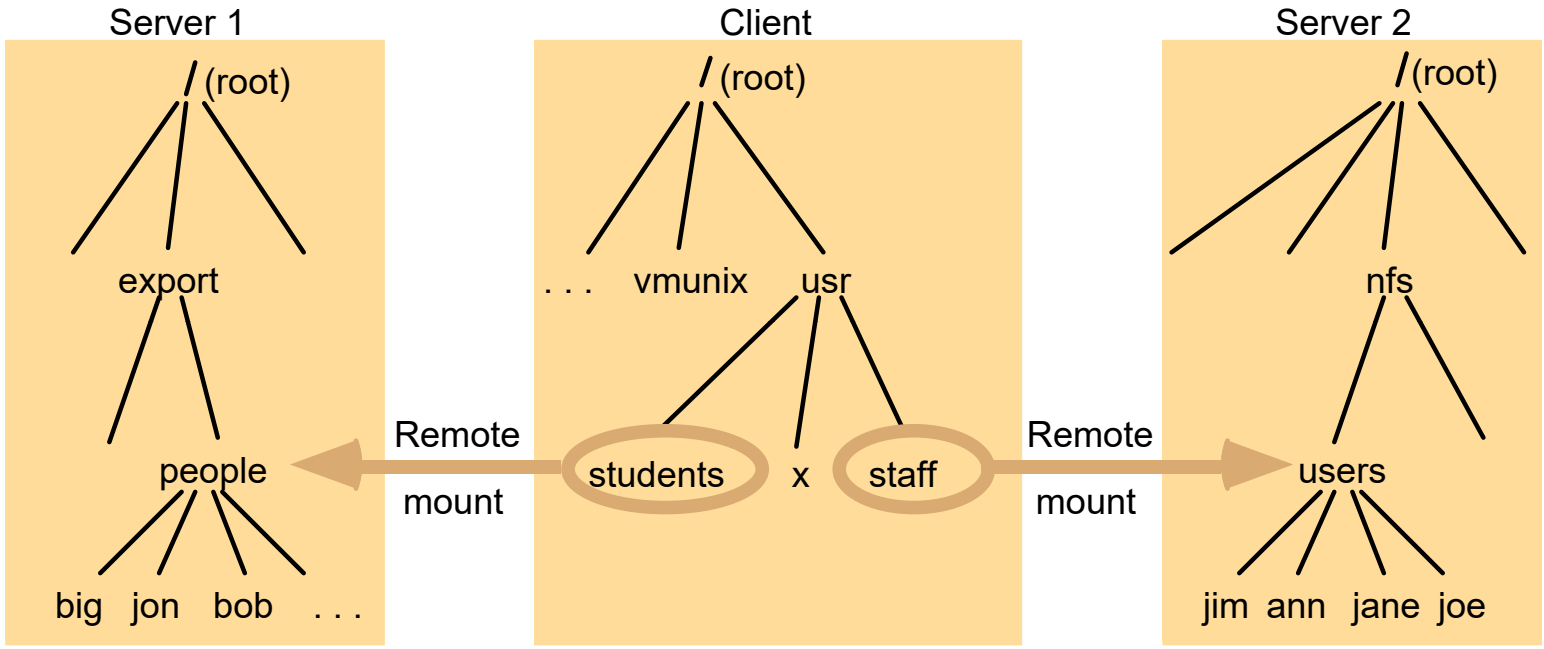
lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

NFS Server Operations Simplified (2)

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Local and Remote File Systems Access

- Accessible from an NFS client
  - The file system mounted at /usr/students in the client is actually the sub-tree located at /export/people in Server 1
  - The file system mounted at /usr/staff in the client is actually the sub-tree located at /nfs/users in Server 2



NFS Concerns

- Auto-mounter
  - Mount on access of any non-mounted file
    - Valid for a set of registered directories
- Server caching
  - Conventional UNIX: read-ahead and delayed write
  - NFS: write-through
- Client caching
  - Timeouts
    - Cache valid iff (T-Tc < c) or (FileAttrclient = FileAttrserver)
- Security
  - Quite flaky → Better with access control systems (Kerberos)

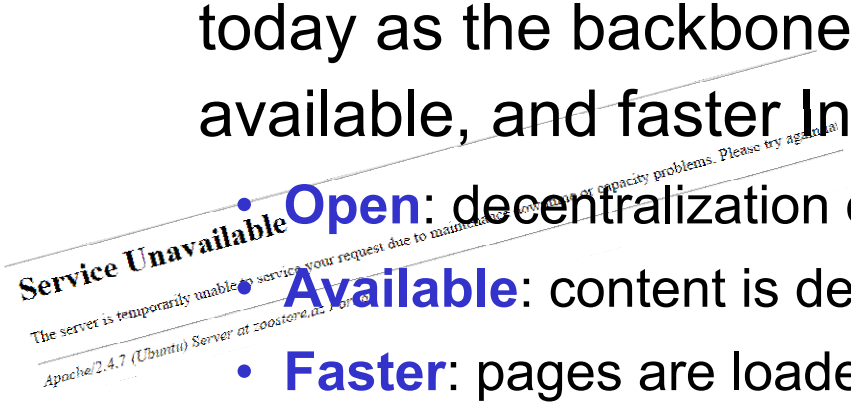
Case Study 1: Summary

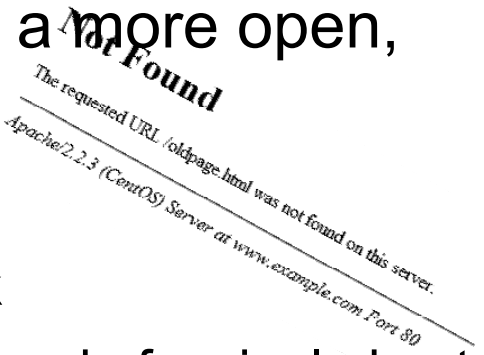
- Transparency
  - Access — NFS client process
  - Location — File name space identical
  - Mobility — Remounts possible
  - Scaling — Only if no “hot spot files”
- Concurrent file updates — Not supported
- File replication — Read-only files
- Hardware and software heterogeneity — Given
- Fault tolerance — Stateless and idempotent
- Consistency — Close to one-copy semantics
- Security — Medium
- Efficiency — Seems ok (still in larger use)

Case Study 2: IPFS

- Inter Planetary File System
  - Defined as a protocol for the file system defining a decentralized storage and delivery network build upon peer-to-peer (p2p) and content-addressed file-system principles

- Goal
  - Replacing centralized systems and protocols, which serve today as the backbone of the Web providing a more open, available, and faster Internet
    - **Open**: decentralization of services and content
    - **Available**: content is decentralized in the network
    - **Faster**: pages are loaded from multiple peers instead of a single host

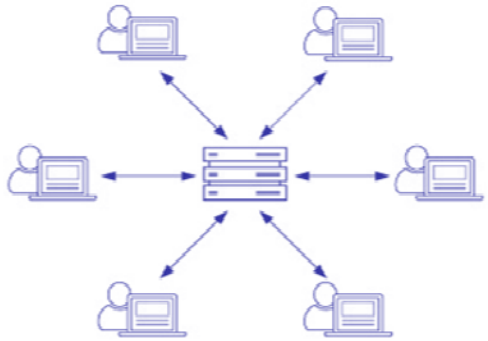




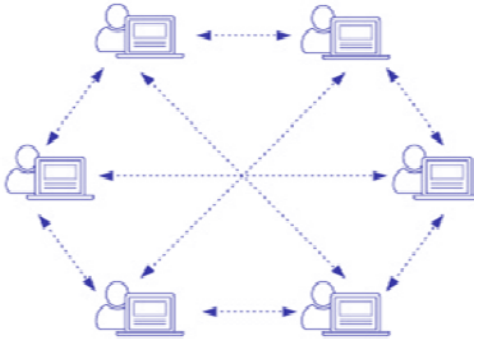
Contrasting Juxtaposition: HTTP and IPFS

- Hyper Text Transfer Protocol (HTTP)
  - Backbone protocol of the Internet
  - Essentially centralized implemented
    - Relying on hosts to provide content/services
    - Widely deployed
- IPFS
  - Essentially decentralized
    - Reliably persisting characteristics
    - Secured against DDoS attacks
    - Reduced time to load webpages
  - Drawbacks
    - Hard maintenance and privacy concerns

HTTP Model



IPFS Model



DDoS: Distributed Denial-of-Service Attacks

Feature-based Comparison: HTTP and IPFS

Feature	IPFS	HTTP
Architecture	Decentralized	Centralized
Addressing	**Content-based** whereas files are identified by a hash	Location-based whereas hosts are identified by an IP address
Bandwidth	**Higher** as data is retrieved from multiple peers	**Lower** since data retrieval' relies on host's upload bandwidth
Availability	**Higher** since data is decentralized across multiple peers	**Lower** since data is stored in a single host
Privacy	**Lower** since data is decentralized across multiple unknown peers	**Higher** since data is stored at known hosts
Adoption	**Slower** since it is a relatively new protocol	**Established** and industry-standard protocol
Costs	**Lower** since every peer can host content	**Higher** as a single host should provide the entire content/service

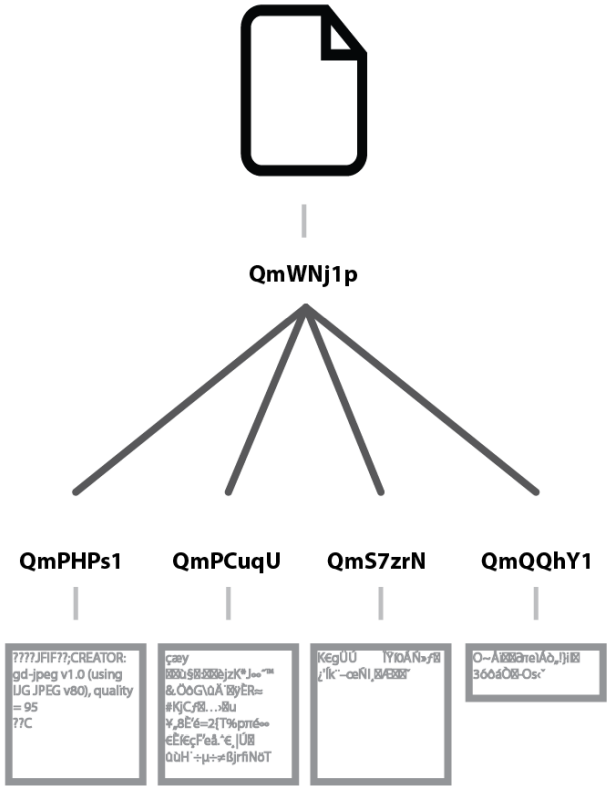
IPFS Building Blocks

- IPFS as a distributed file system seeks to connect all computing devices with the same system of files
- IPFS built upon major components
  A. Merkle DAG: uses a Merkle Tree or a Merkle DAG similar to the one used in the Git version control system
  B. Block Exchange and Network: used in the BitTorrent Protocol (also known as BitSwap) to exchange data between nodes, it is a p2p file sharing protocol, which coordinates data exchange between untrusted swarms
  C. Distributed Hash Tables (DHT): Kademlia is used to store and retrieve data across nodes in the network (*i.e.*, content-based location)

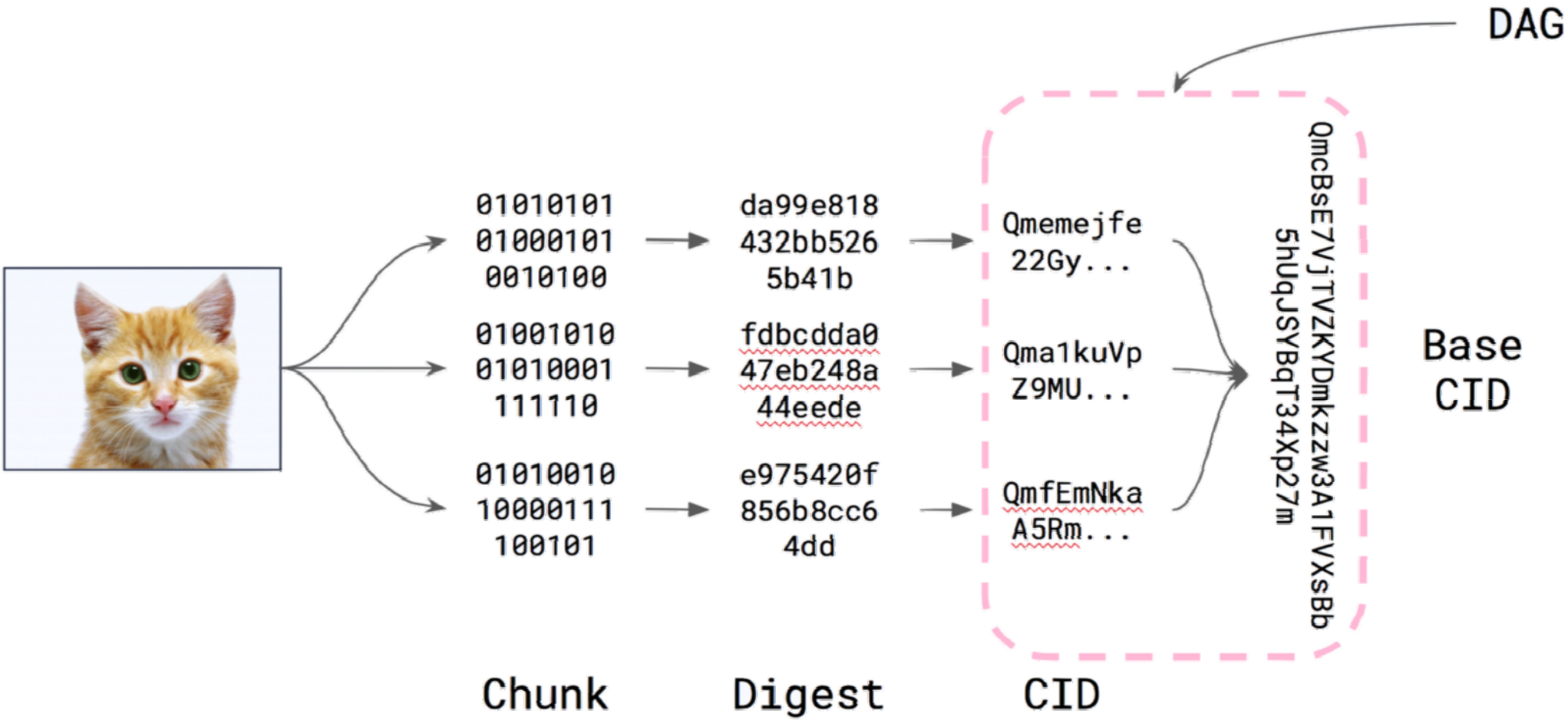
DAG: Directed Acyclic Graph

A. IPFS Merkle DAG (1)

- "Heart" of the IPFS
  - Directed Acyclic Graph (DAG) with links
    - Links are implemented as hashes
  - Properties
    - Authenticated
      - Content can be hashed and verified against the link
    - Permanent
      - Once fetched, objects can be cached forever
    - Universal
      - Any data structure can be represented as a Merkle DAG
    - Decentralized
      - Objects can be created by anyone, without centralized writers

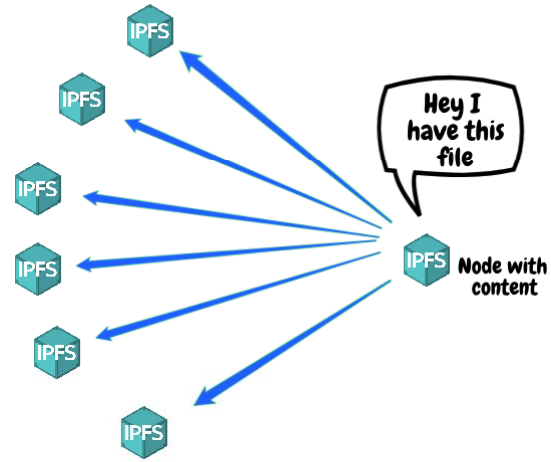


A. IPFS Merkle DAG (2)



B. Block and Node

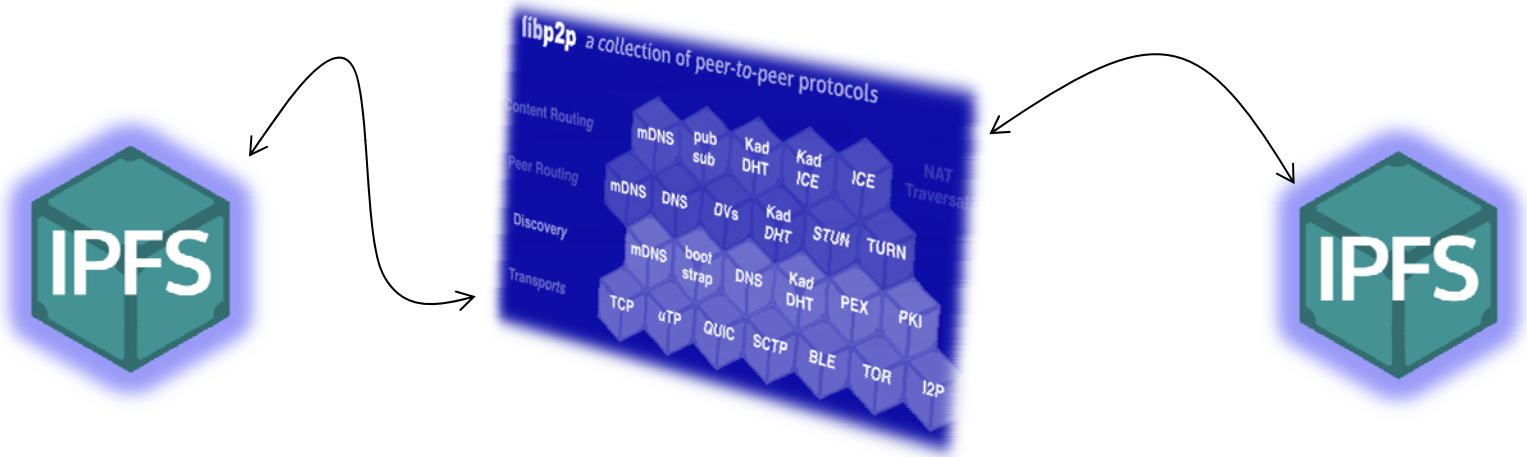
- Blocks represent data to be stored
- IPFS node is a component to find, publish, and replicate "merkledag" objects
  1. "IPFS Block Exchange" negotiates bulk data transfers
     - Bitswap is the main protocol for exchanging data
     - Generalization of BitTorrent to work with arbitrary and not known a priori DAGs
  2. HTTP as a simple exchange implemented with HTTP clients and servers
     - Compatibility reasons



- Protocol to handle object's operations b/w IPFS nodes
  - File sharing protocol coordinates data exchange

B. Block Exchange and Network

- IPFS' underlying network provides
  - Point-to-point transport connections (reliable and unreliable) between any two IPFS nodes
- Protocol needed to exchange data (blocks) between IPFS nodes
  - Based on Bitswap (block exchange) and on libp2p (routing)



B. Routing

- IPFS Routing
  - Protocol defined within a separate layer
    - Peer Routing to find other nodes
    - Content Routing to find data published to IPFS

- IFPS Routing System
  - Defined as an interface
  - Satisfied by different kind of implementations
    - DHT (Distributed Hash Table): used to create a semi-persistent routing record for a distributed cache in the network
    - MDNS (Multicast DNS): used to find services advertised locally
    - DNS (Domain Name System): used to find a node for the storing and sharing of data, can utilize the IPNS (Inter Planetary Name Space)

C. Distributed Hash Table (DHT)

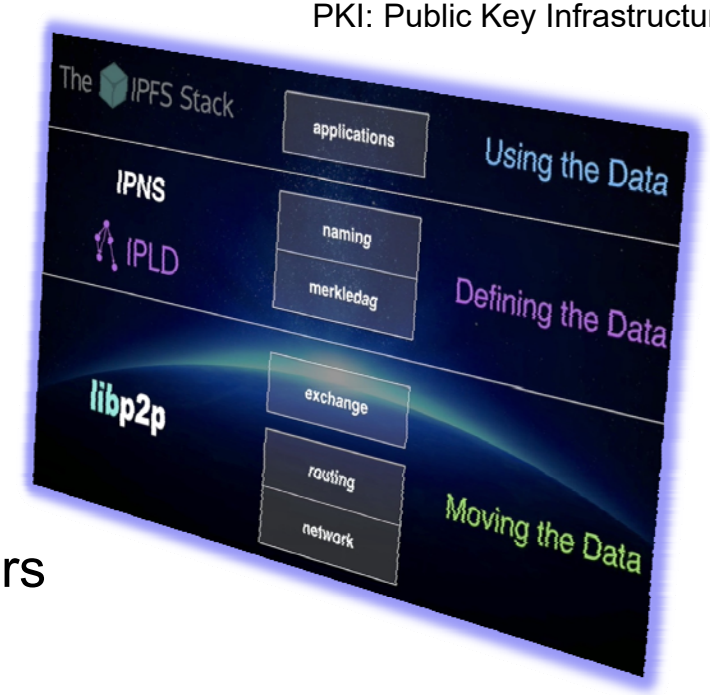
- DHT used as fundamental IPFS component
  - For the routing system
  - Acts like a cross between a catalog and a navigation system

Type	Purpose	Used by
Provider records	Map a data identifier (*i.e.*, a multi-hash) to a peer that has advertised that they have that content and are willing to provide it to you.	- IPFS to find content - IPNS over PubSub to find other members of the pubsub *topic*.
IPNS records	Map an IPNS key (i.e., the hash of a public key) to an IPNS record (i.e., a signed and versioned pointer to a path like /ipfs/bafyxyz...)	- IPNS
Peer records	Map a peerID to a set of multia-ddresses at which the peer may be reached	- IPFS when we know of a peer with content, but do not know its address. - Manual connections (*e.g.*, ipfs swarm connect /p2p/Qmxyz...)

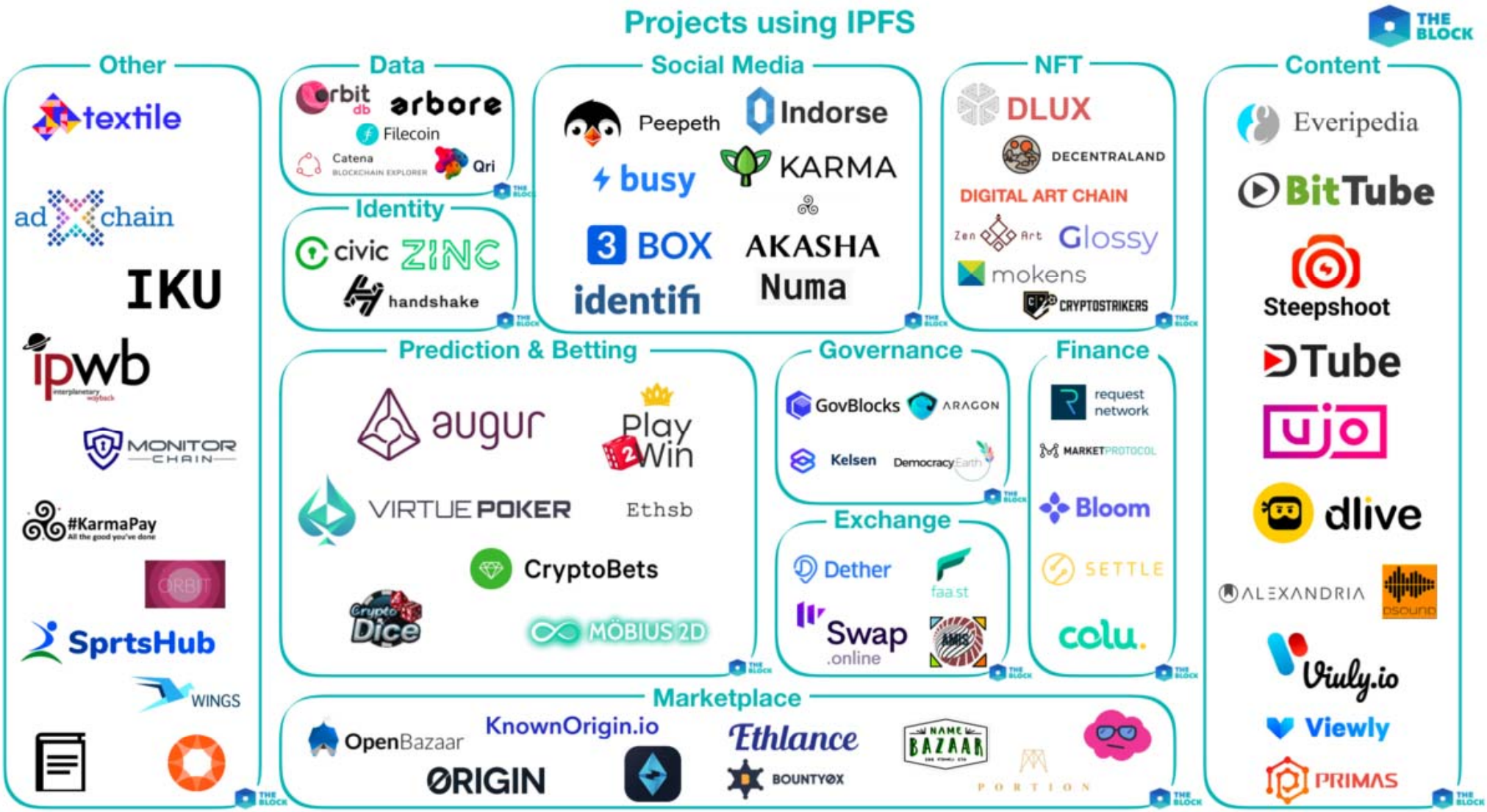
IPFS Protocol Stack

- IPFS protocol stack offers modular protocols
  - Each layer may have multiple implementations
  - All may use different modules
    - Applications: Web apps, Blockchain
    - Naming: self-certifying PKI for IPNS
    - MerkleDAG:
      data structure format (thin waist)
    - Exchange:
      block transport and replication
    - Routing: locating peers and objects
    - Network:
      establishing connections between peers

PKI: Public Key Infrastructure



IFPS Applications and Projects



IPFS' Privacy

- IPFS utilizes a public network – the Internet
- Methods to ensure privacy are (partially) essential
  - Private IPFS Network
    - Behaves similarly to the public network except for the fact that participants are only able to communicate with other nodes inside that same private IPFS network
  - Content Encryption
    - Considering encryption of content uploaded to the IPFS network,
    - It still can be tracked, but content remains "unreadable"
  - Gateway Usage
    - Requesting content through a public gateway allows a user to retrieve content from the IPFS network without running their own node
    - Hiding identities behind gateways

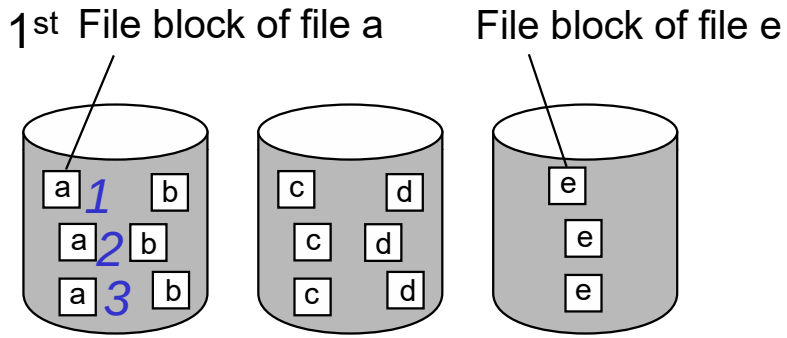
Case Study 2: Summary

- IPFS defines a system and related protocols to decentralize services and content offered on the Internet
  - Increased performance and availability due to its decentralized nature, but not “private”
  - On-going project with “complex” configurations for daily users

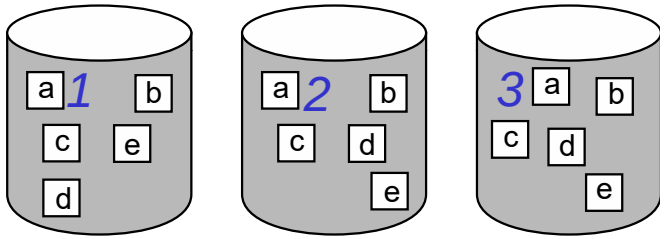
- Growing list of use cases and applications
  - Standalone and decentralized file system
  - Complementary decentralized storage to existing HTTP-based and centralized systems
    - Important role as decentralized storage for Blockchains

Case Study 3: Google File System

- Designed for large files and appends to files
- Files are distributed over multiple servers
  in chunks of 64 MB (like in a RAID) RAID: Redundant Array of Independent Disks
  → File striping
- Chunks are replicated (not shown below)



Each file on one disc



Each file on several discs

Google File System Architecture

- Master serves acts as directory only
- All chunks are distributed over chunk server

  → better scalability