Distributed and Federated Storage

How to store things… in… many places... (maybe)

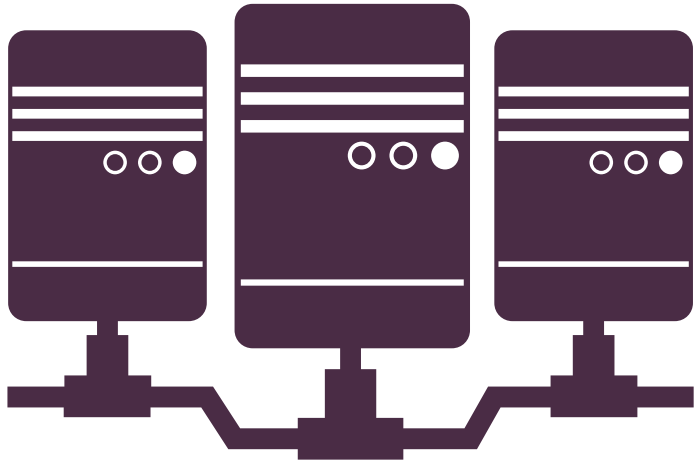
CS2510

Presented by: **wilkie**
dwilk@cs.pitt.edu

University of Pittsburgh

Recommended Reading (or Skimming)

- NFS: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.473
- WAFL: https://dl.acm.org/citation.cfm?id=1267093
- ***Hierarchical File Systems are Dead*** (Margo Seltzer, 2009): https://www.eecs.harvard.edu/margo/papers/hotos09/paper.pdf
- Chord (Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, 2001): https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
- ***Kademlia*** (Petar Maymounkov, David Mazières, 2002): https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf
- BitTorrent Overview: http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf
- ***IPFS*** (Juan Benet, 2014): https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf (served via IPFS, neat)



Network File System

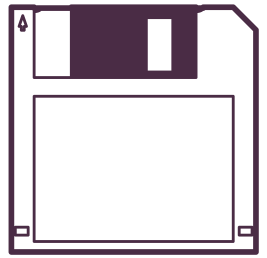
NFS: A Traditional and Classic Distributed File System

Problem

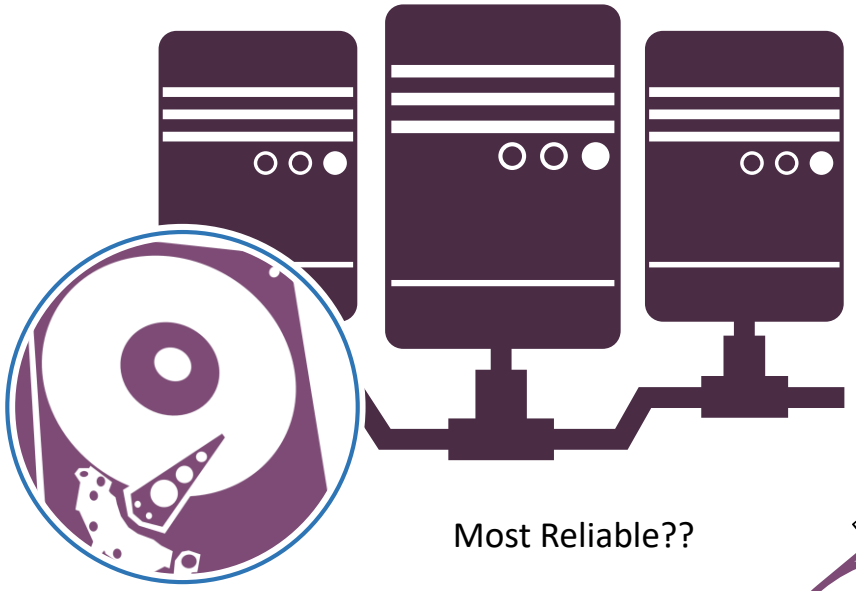
- Storage is cheap.
  - YES. This is a *problem* in a classical sense.
  - People are storing more stuff and want very strong storage guarantees.
  - Networked (web) applications are global and people want strong availability and stable speed/performance (wherever in the world they are.) *Yikes!*
- More data == Greater probability of failure
  - We want **consistency** (correct, up-to-date data)
  - We want **availability** (when we need it)
  - We want **partition tolerance** (even in the presence of downtime)
  - Oh. Hmm. Well, heck.
  - That's hard (technically impossible) so what can we do?

Lightning Round: Distributed Storage

- Network File System (NFS)
  - We will gloss over details, here, but the papers are definitely worth a read.
  - It invented the Virtual File System (VFS)
  - Basically, though, it is an early attempt to investigate the ***trade-offs*** for client/server file consistency

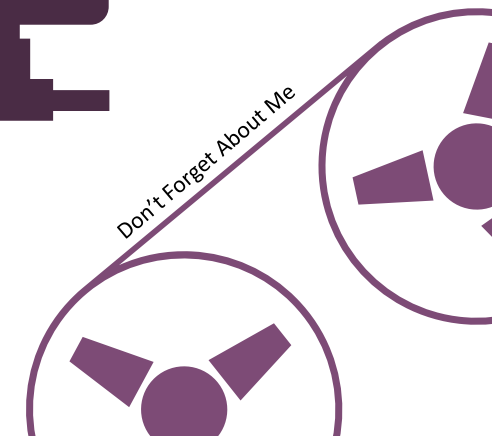


Unreliable



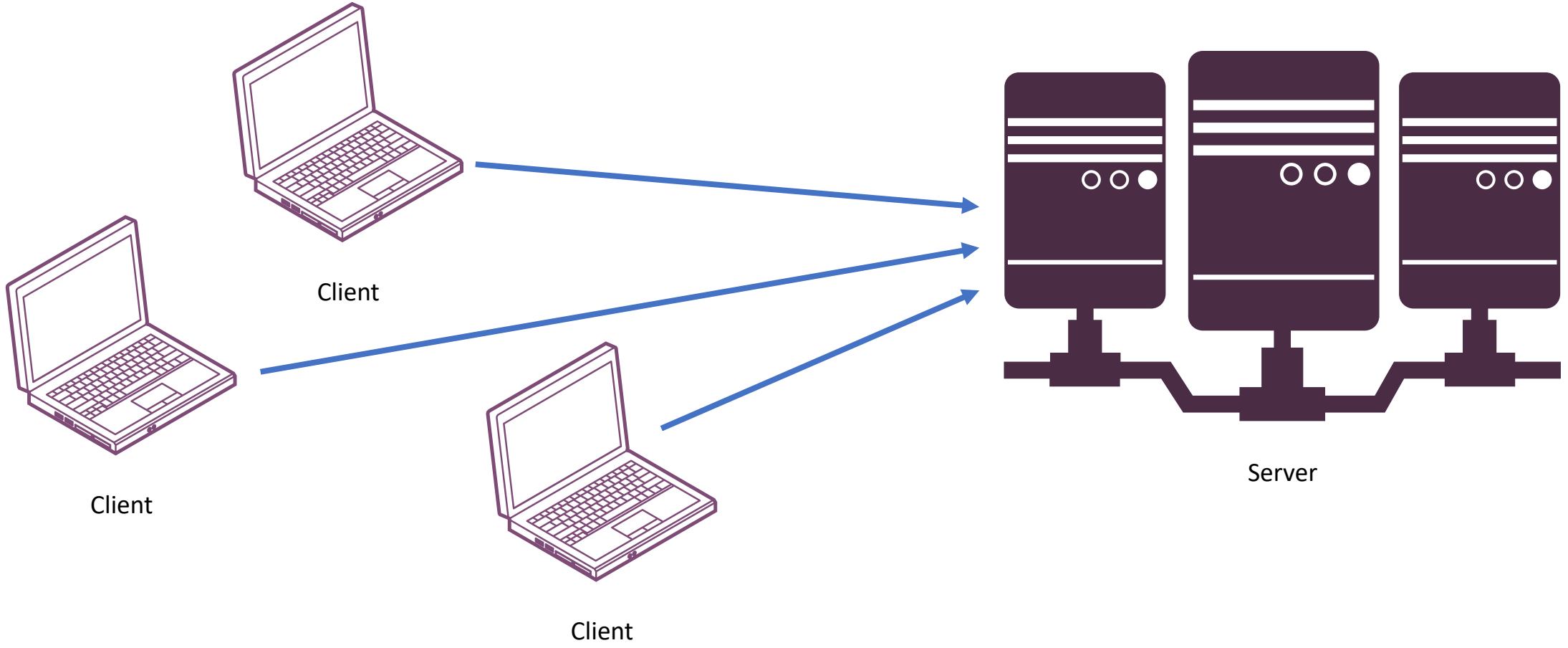
Most Reliable??

There's more to it obviously…..



NFS System Model

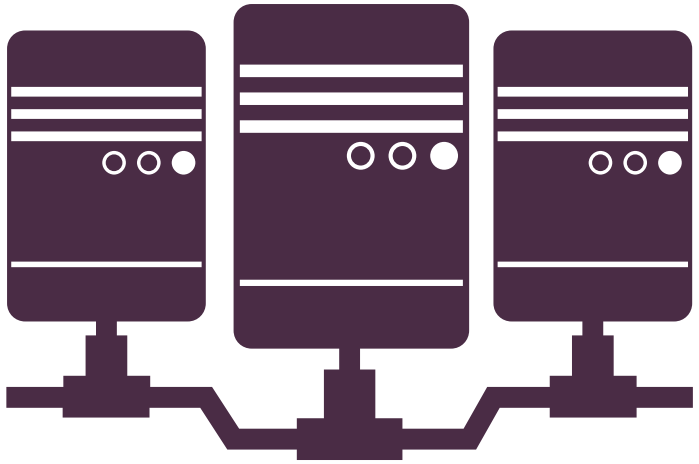
- Each client connects directly to the server. Files could be duplicated on client-side.



NFS Stateless Protocol

Set of common operations clients can issue: *(where is open? close?)*

lookup	Returns file handle for filename
create	Create a new file and return handle
remove	Removes a file from a directory
getattr	Returns file attributes (stat)
setattr	Sets file attributes
read	Reads bytes from file
write	Writes bytes to file



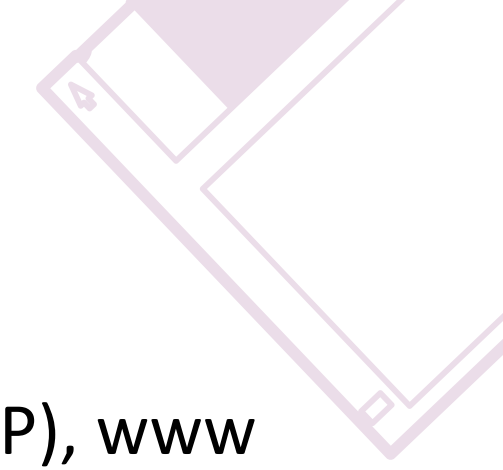
Commands sent to the server. (one-way)

Statelessness (Toward Availability)

- NFS implemented an **open** (standard, well-known) and **stateless** (all actions/commands are independent) protocol.

- The **open()** system call is an example of a *stateful protocol*.
  - The system call looks up a file by a path.
  - It gives you a file handle (or file pointer) that represents that file.
  - You give that file handle to **read** or **write** calls. (not the path)
  - The file handle does not directly relate to the file. (A second call to open gives a different file handle)
  - If your machine loses power… that handle is lost… you'll need to call **open** again.

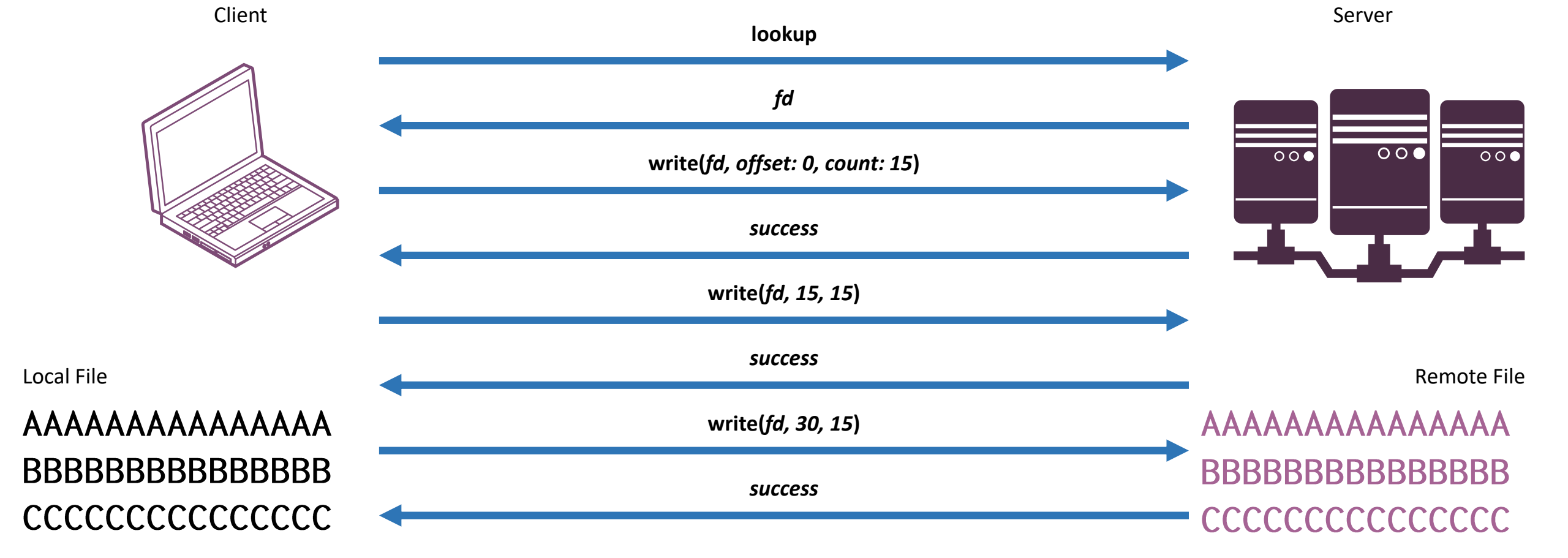
Statelessness (Toward Availability)



- Other stateless protocols: HTTP (but not FTP), IP (but not TCP), www
- So, in NFS, we don't have an **open**.
- Instead we have an ***idempotent*** lookup function.
  - Always gives us a predictable file handle. Even if the server crashes and reboots.
- Statelessness also benefits from *idempotent* read/write functions.
  - Sending the same write command twice in a row shouldn't matter.
- This means ambiguity of server crashes (did it do the thing I wanted?) doesn't matter. Just send the command again. No big deal. *(kinda)*
  - NFS's way of handling **duplicate requests**. (See Fault Tolerance slides)
- ***Consider***: What about mutual exclusion?? (file locking) Tricky!

Statelessness And Failure (NFS) [best]

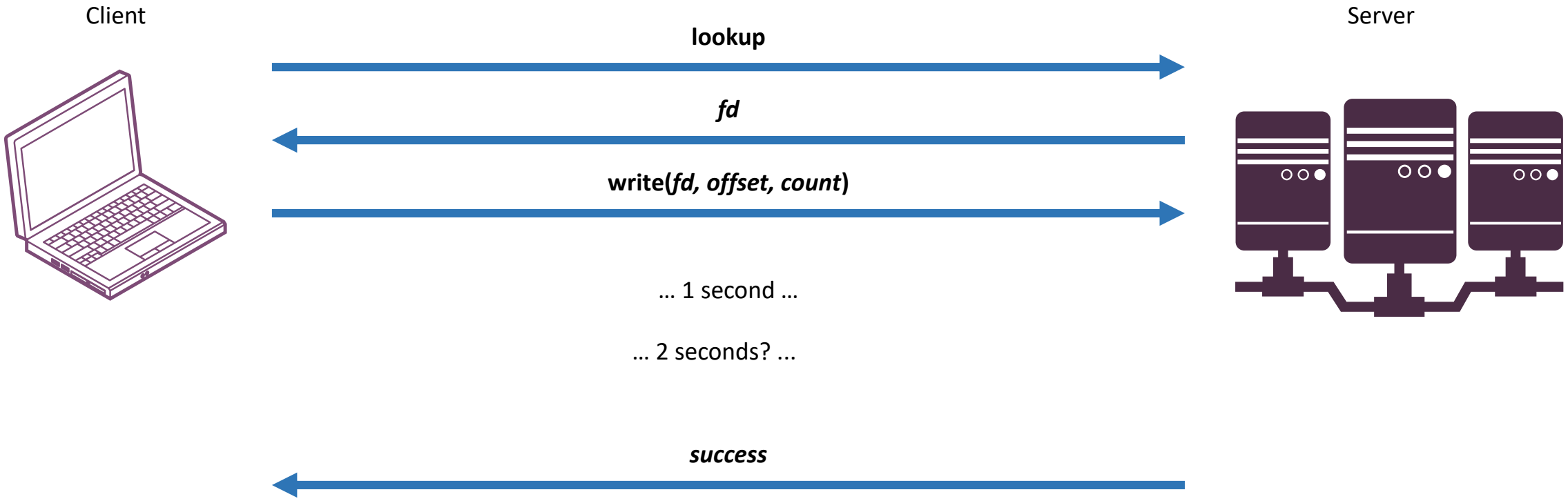
A client issues a series of writes to a file located on a particular server.



Server-side Writes Are Slow

Problem: Writes are really slow…

(Did the server crash?? Should I try again?? Delay… delay… delay)

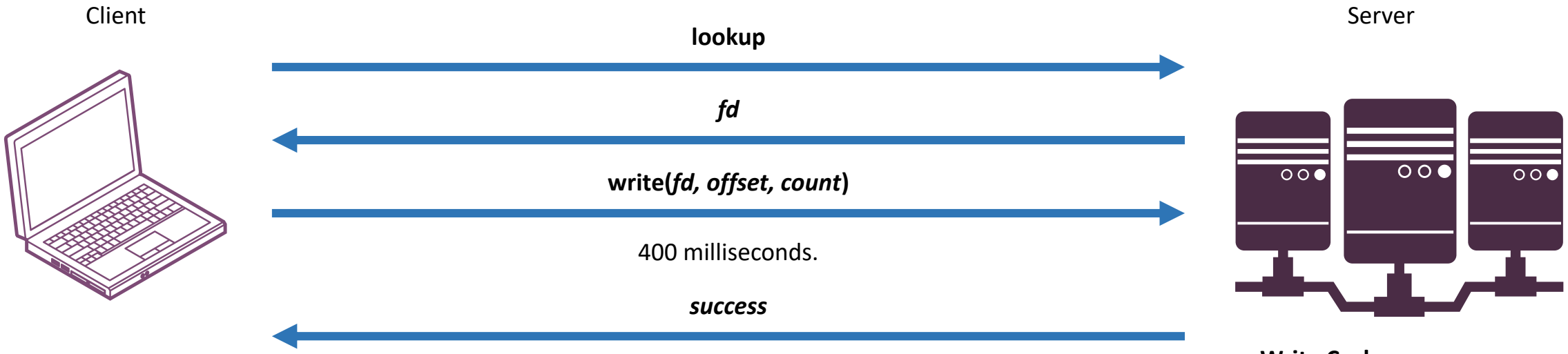


Time relates to the amount of data we want to write***… is there a good block size?***
1KiB? 4KiB? 1MiB? (bigger == slower, harsher failures; small == faster, but more messages)

Server-side Write Cache?

Solution: Cache writes and commit them when we have time.

(Client gets a respond much more quickly… but at what cost? There's always a ***trade-off***)



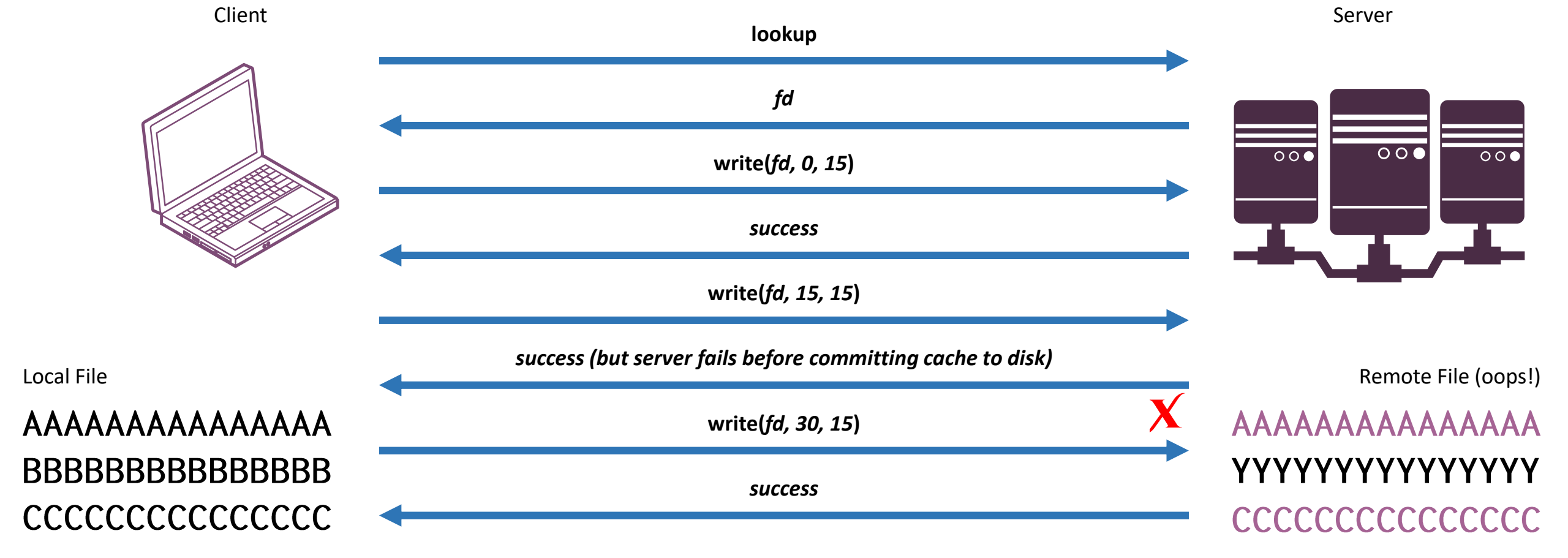
Write Cache:
Need to write this block at some point!

But what if… ***it doesn't?***

When should it write it back? Hmm. It is not that obvious.
(Refer to Consistency discussion from previous lectures)

Write Cache Failure (NFS)

A server **must** commit changes to disk if it tells client it succeeded…
If it *did fail*, and restarted quickly, the client would never know!



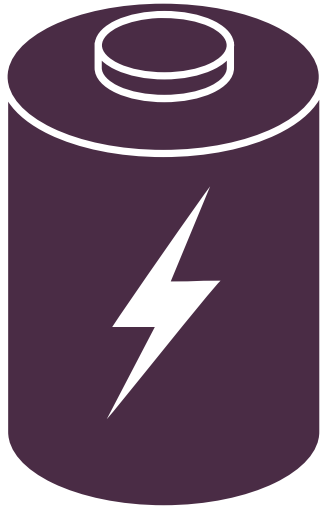
Fault Tolerance

- So, we can allow failure, but *only if we know if an operation succeeded*. (we are assuming a strong eventual consistency)
  - In this case, writes… but those are really slow. Hmm.
- Hey! We've seen this all before…
  - This is all fault tolerance basics.
  - But this is our chance to see it in practice.
- [a basic conforming implementation of] NFS makes a ***trade-off***. It gives you distributed data that is reliably stored at the cost of slow writes.

- Can we speed that up?

Strategies

- Problem: Slow to send data since we must wait for it to be committed.
  - Also, we may write (and overwrite) data repeatedly.
  - How to mitigate performance?
- Possibility: Send writes in smaller chunks.
  - Trade-offs: More messages to/from server.
- Possibility: We can cache writes at the client side.
  - Trade-offs:
    - Client side may crash.
    - Accumulated writes may stall as we send more data at once.
    - Overall difficulty in knowing when we writeback.



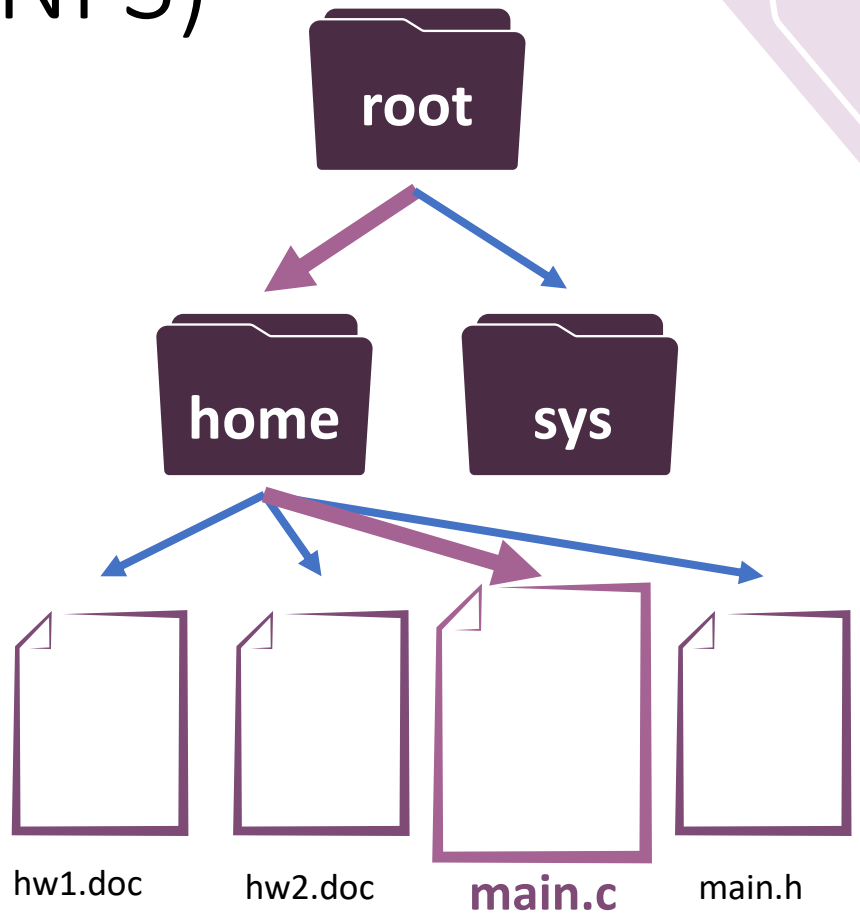
- Possibility: We mitigate likelihood of failure on server.
  - Battery-backed cache, etc. Not perfect, but removes client burden.
  - Make disks faster (Just make them as fast as RAM, right? NVRAM?) ☺
  - Distribute writeback data to more than one server. (partitioning! Peer-to-peer!!)

File System Structure

From Classic Hierarchical to Non-Traditional

File System Layout (Classical; NFS)

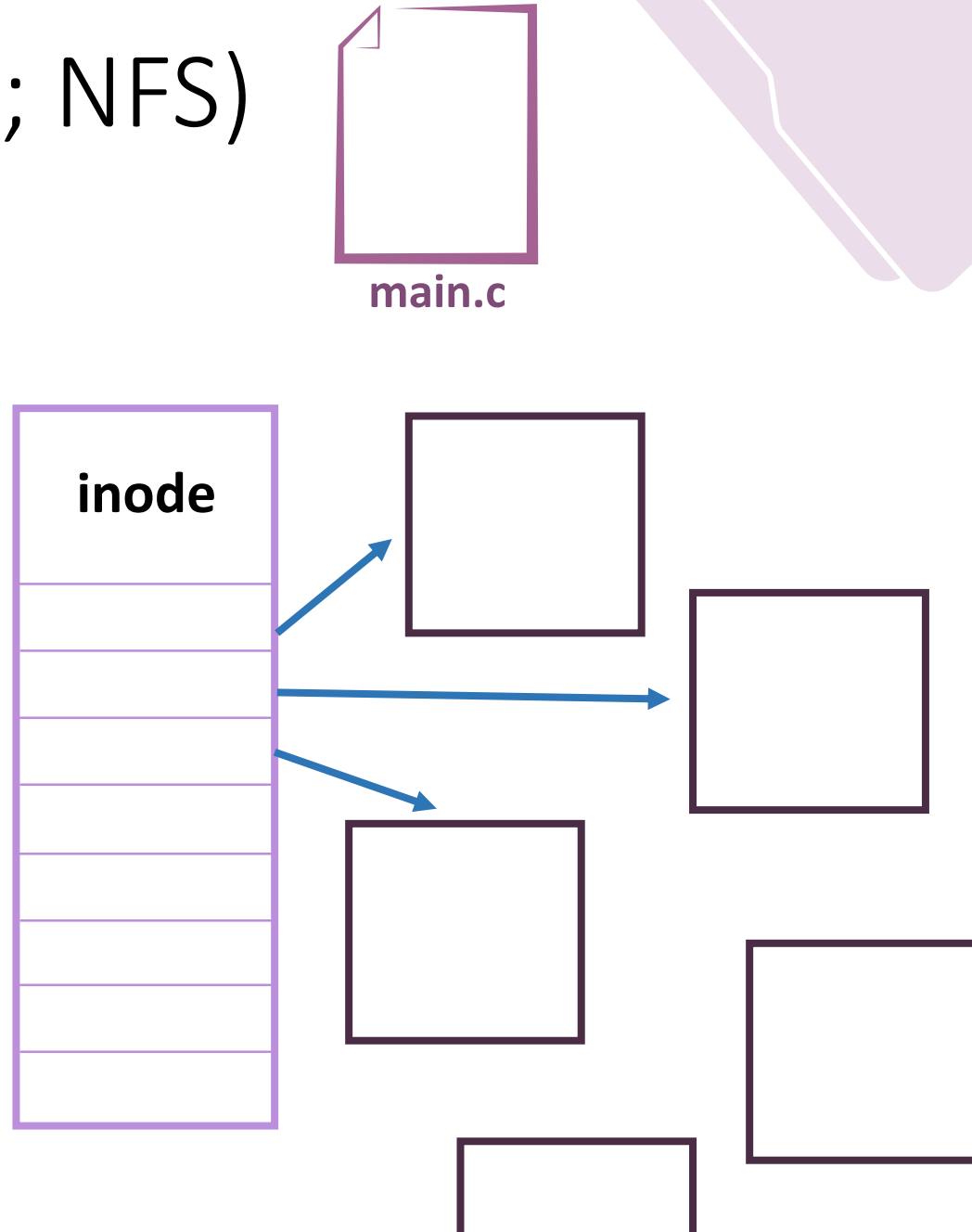
- We generally are used to a very classical layout: directories and files.
- NFS introduced the Virtual File System, so some directories could be ***mounted*** as remote (or devices)
  - Therefore, some file paths have more latency than others! Interesting.
- We navigate via a path that strictly relates to the layout of directories as a tree. (***Hierarchical Layout***)



/root/home/main.c

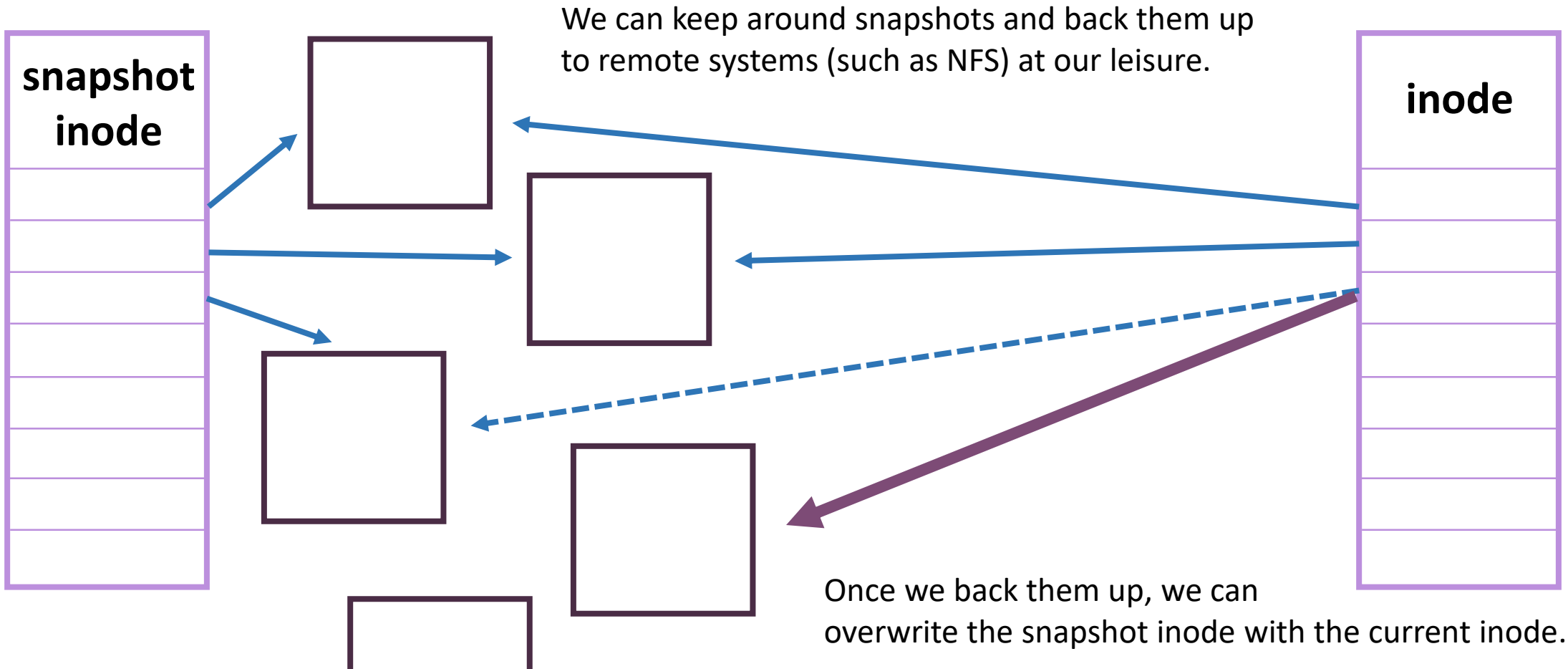
File System Layout (Classical; NFS)

- This should be CS1550-ish OS review!
- Files are broken down into inodes that point to file data. (indirection)
- An inode is a set of pointers to blocks on disk. (it may need inodes that point to inodes to keep block sizes small)
- The smaller the block size, the more metadata (inodes) required.
  - But easier to backup what changes.
  - (We'll see why in a minute)



Cheap Versioning (WAFL+NFS)

- Simply keep copies of prior inodes to maintain a simple snapshot!



Directories and Hierarchies

- Hierarchical directories are based on older types of computers and operating systems designed around severe limitations.
- NFS (+VFS) mounts remote servers to directories.
- This is convenient (easy to understand and configure) for smaller storage networks.
- However, two *different* files may have the same name and exist on two different machines.
  - How to differentiate? How to find what you want?

Reconsidering Normal (Name-Addressed)

- Currently, many everyday file systems haven't changed much.
  - They are **name-addressed**, that is, you look them up by their name.
- File lookups in hierarchies require many reads from disparate parts of disk as you open and read metadata for each directory.
  - This can be slow. OSes have heavy complexity and caching for directories.
  - Now, consider distributed file systems… if directories span machines!
- There are other approaches. Margo Seltzer in *Hierarchical File Systems are Dead* suggests a tag-based approach more in line with databases: offering indexing and search instead of file paths.

Content Addressing

- However, one approach “flips the script” and allows file lookups to be done on the *data of the file*.
- That seems counter-intuitive: looking up a file via a representation of its data. How do you know the data *beforehand*?

- With **content-addressing**, the file is stored with a name that is derived mathematically from its data as a hash. (md5, sha, etc)
- That yields many interesting properties we will take advantage of.

Hash Function Overview

Good Hash Functions:

- Are one-way (non-invertible)
  - Cannot compute original x from result of $hash(x)$
- Are deterministic
  - $hash(x)$ is equal to $hash(x)$ at any time on any other machine
- Are uniform
  - Are hashes have equal probability. That is:
  - The set H defined by taking a random set and applying $hash(x)$ results in a normal distribution.
- Continuous
  - Hashing two similar numbers should result in a dramatically different hash.
  - That is: $hash(x)$ should be unpredictably distant from $hash(x + 1)$

Basic Hashing

- For simple integrity, we can simply hash the file.
  $k = hash(file)$ is generated. Then key k can be used to open the file.

- When distributing the file, one can know it got the file by simply hashing what it received.
  - Since our hash function is ***deterministic*** the hash will be the same.
  - ***If it isn't, our file is corrupted.***

- In digital archival circles, this is called ***fixity***.

Chunking

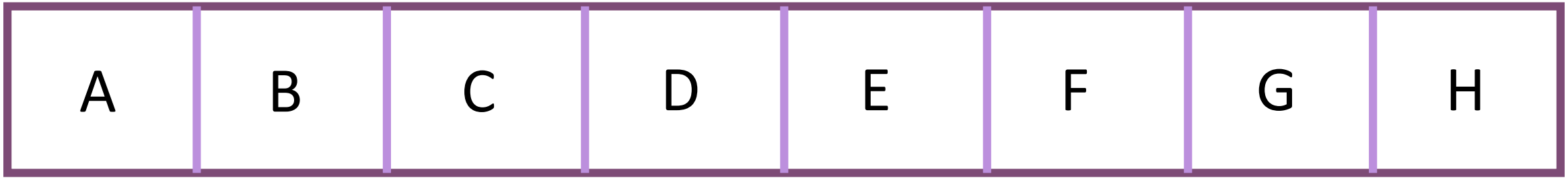
- However, it would be nice to determine which *part* of the file was distributed incorrectly.
  - Maybe we can ask a different source for just that part.
    - Hmm… that's an idea! (we'll get there)

- Dividing up the file is called ***chunking***, and there are things to consider: (*trade-offs!*)
  - How big are the chunks… the more chunks, the more hashes; the more metadata!
  - Of course, the more chunks, the smaller the chunk; therefore, the less window for detecting corruption!

Chunking

- Take a file, divide it into chunks, hash each chunk.

vacation_video.mov




```
A = 912ec803b2ce49e4a541068d495ab570
B = 277f255555a1e4ff124bdacc528b815d
C = 0bdba65117548964bad7181a1a9f99e4
D = 495aa31ae809642160e38868adc7ee8e
E = 23c82b0ba3405d4c15aa85d2190e2cf0
F = b2e7af8aff7c2dd98536ce145d705e7f
G = ce3c4edbce0b4da2d9369e8d14e7677a
H = 93ab352ffd32037684257b39eddf33dd
```

Distribution (Detecting Failure)

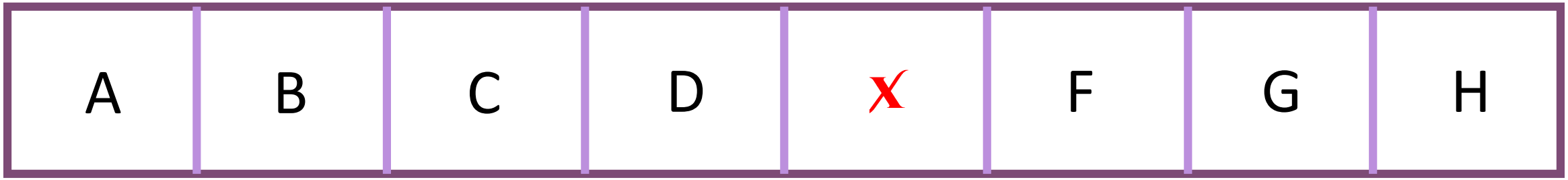
- Client requests the hashes given.

```
A = 912ec803b2ce49e4a541068d495ab570
B = 277f255555a1e4ff124bdacc528b815d
C = 0bdba6511754896 4bad7181a1a9f99e4
D = 495aa31ae809642160e38868adc7ee8e
E = 23c82b0ba3405d4c15aa85d2190e2cf0
F = b2e7af8aff7c2dd98536ce145d705e7f
G = ce3c4edbce0b4da2d9369e8d14e7677a
H = 93ab352ffd32037684257b39eddf33dd
```

But receives chunks with hashes:

```
A’ = 912ec803b2ce49e4a541068d495ab570
B’ = 277f255555a1e4ff124bdacc528b815d
C’ = 0bdba6511754896 4bad7181a1a9f99e4
D’ = 495aa31ae809642160e38868adc7ee8e
E’ = ecf5b19f62a8037f97217ed9cb9b98d9
F’ = b2e7af8aff7c2dd98536ce145d705e7f
G’ = ce3c4edbce0b4da2d9369e8d14e7677a
H’ = 93ab352ffd32037684257b39eddf33dd
```

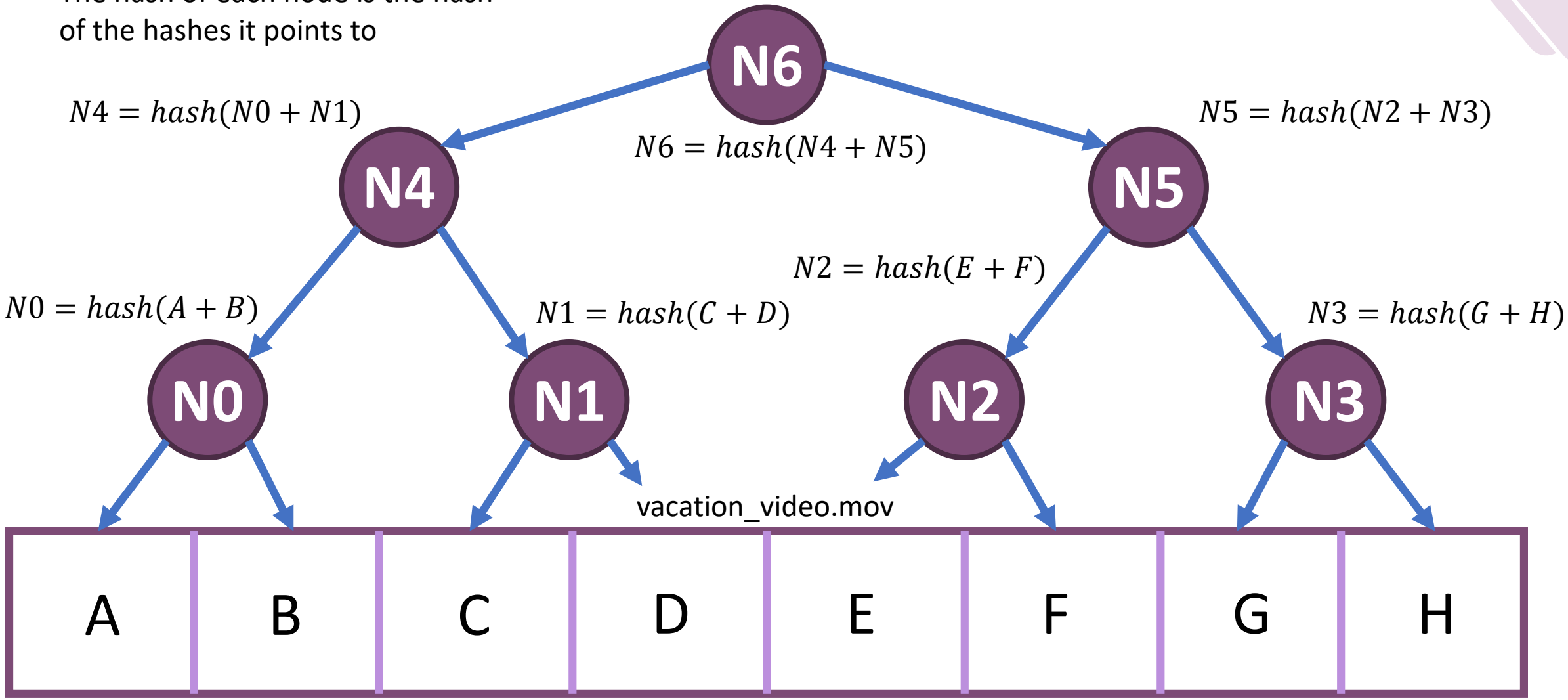
vacation_video.mov



Merkle Tree/DAG

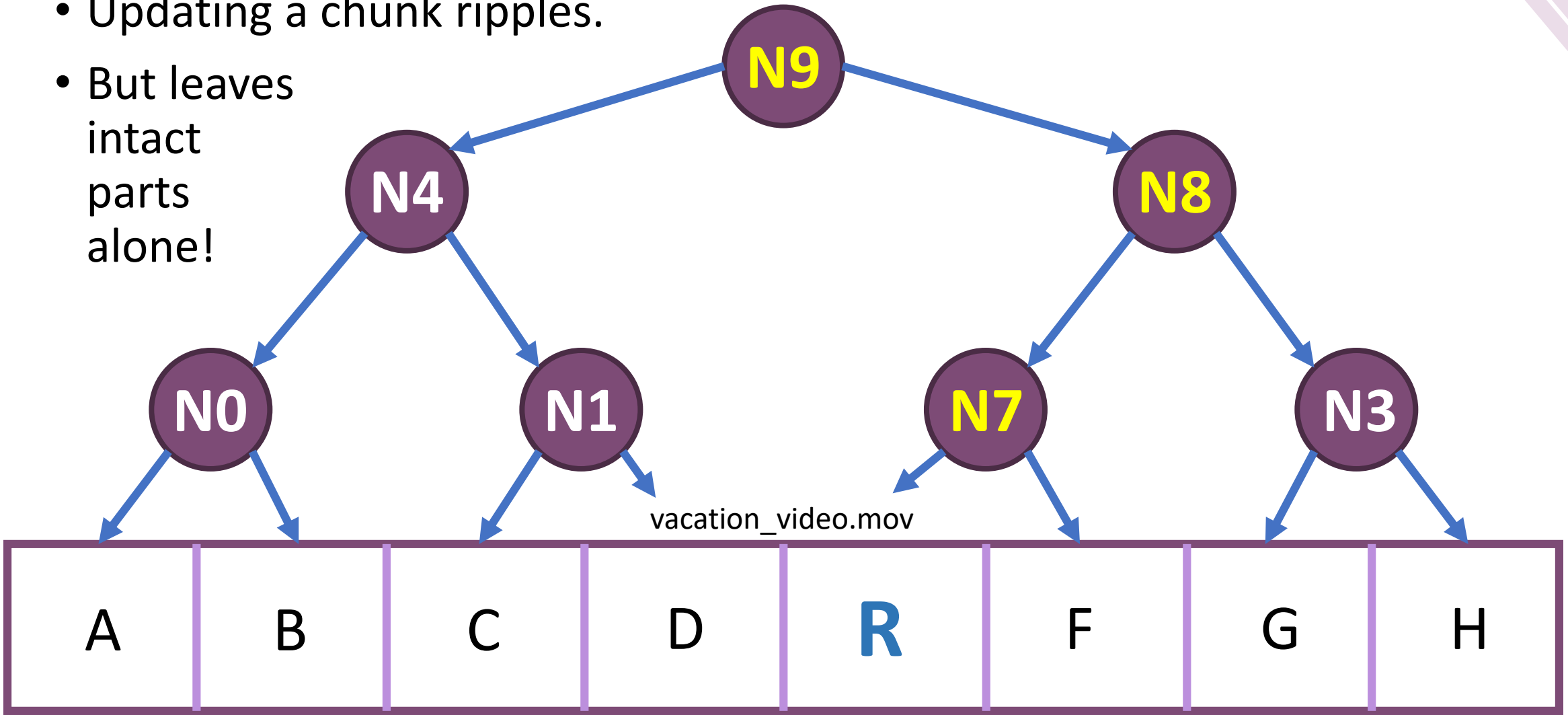
The hash of each node is the hash of the hashes it points to

We can organize a file such that it can be referred to by a single hash, but also be divided up into more easily shared chunks.



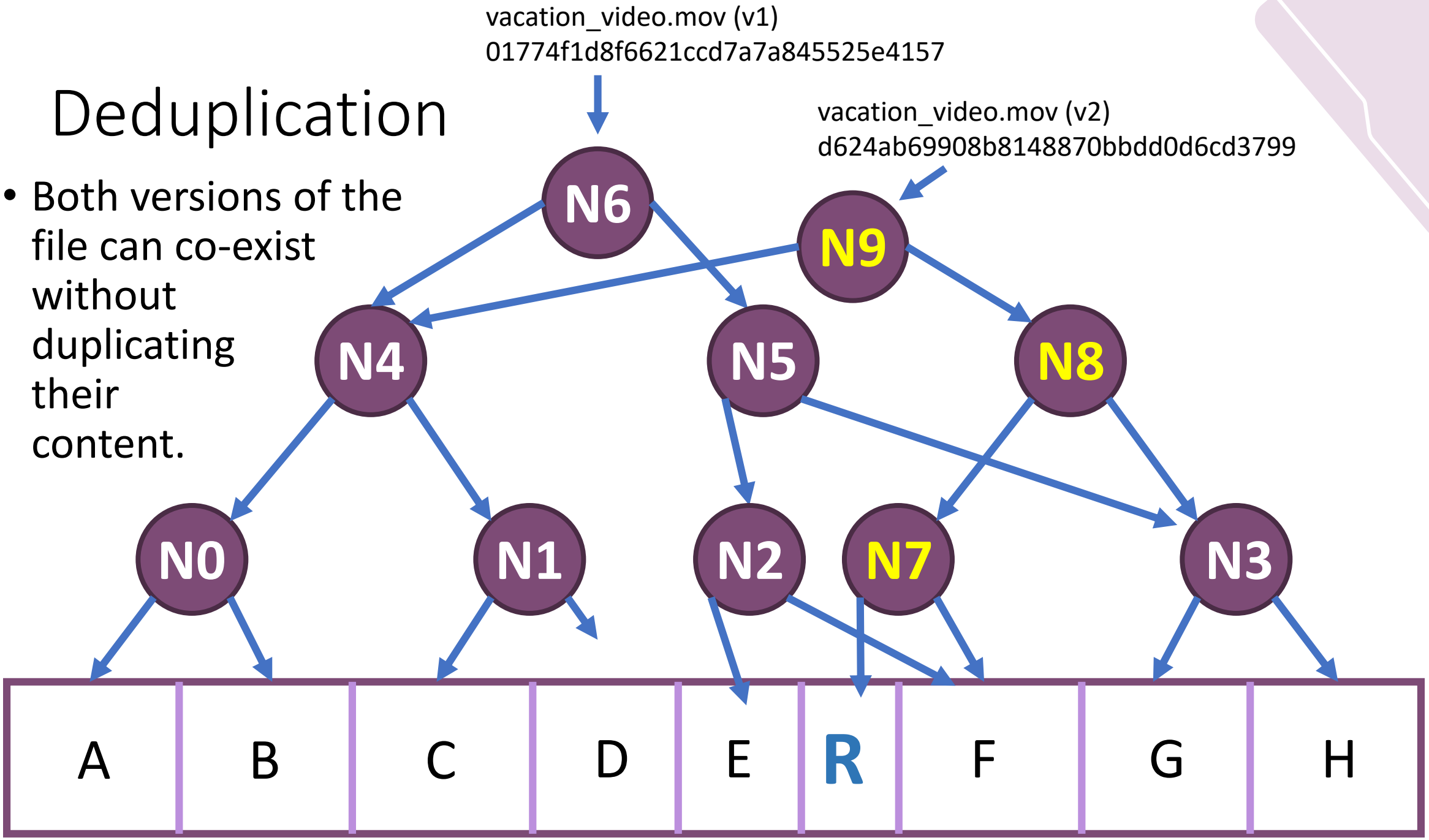
Merkle-based Deduplication

- Updating a chunk ripples.
- But leaves intact parts alone!

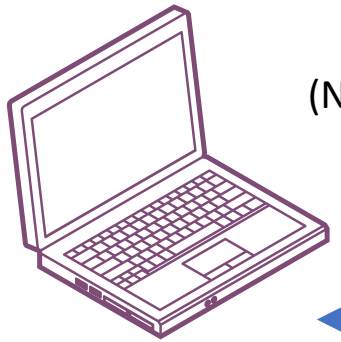


Deduplication

- Both versions of the file can co-exist without duplicating their content.



Distribution



(N1) 01774f1d8f6621ccd7a7a845525e4157

{N4, N5}

(N4) aa7e074434e5ae507ec22f9f1f7df656

{N0, N1}

(N1) aa7e074434e5ae507ec22f9f1f7df656

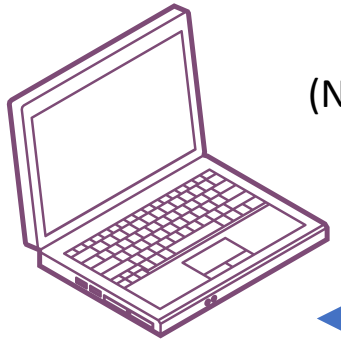
{C, D}

(D) 495aa31ae809642160e38868adc7ee8e

D's File Data

- I can ask a storage server for the file at that hash.
- It will give me the sub hashes.
- At each step, I can verify the information by hashing what I downloaded!

Distribution



(N1) 01774f1d8f6621ccd7a7a845525e4157

{N4, N5}

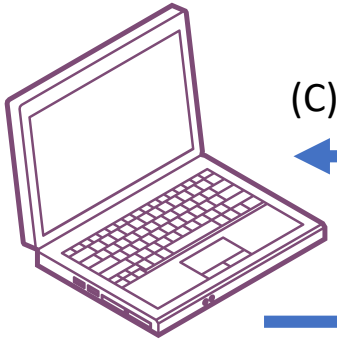
- Nothing is stopping me from asking multiple servers.
- ***But how do I know which servers have which chunk??*** Hmm.

(N4) aa7e074434e5ae507ec22f9f1f7df656

{N0, N1}

(N1) aa7e074434e5ae507ec22f9f1f7df656

{C, D}



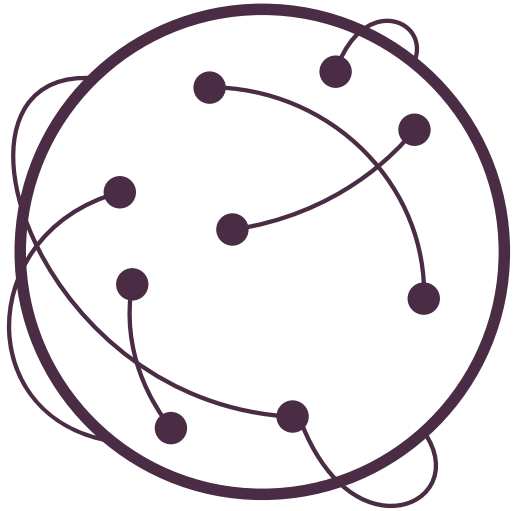
(C) 0bdba65117548964bad7181a1a9f99e4

(D) 495aa31ae809642160e38868adc7ee8e

C's File Data

D's File Data

Concurrently gather two chunks at once!



Peer-to-peer Systems

BitTorrent, Kademlia, and IPFS: Condemned yet Coordinated.

BitTorrent

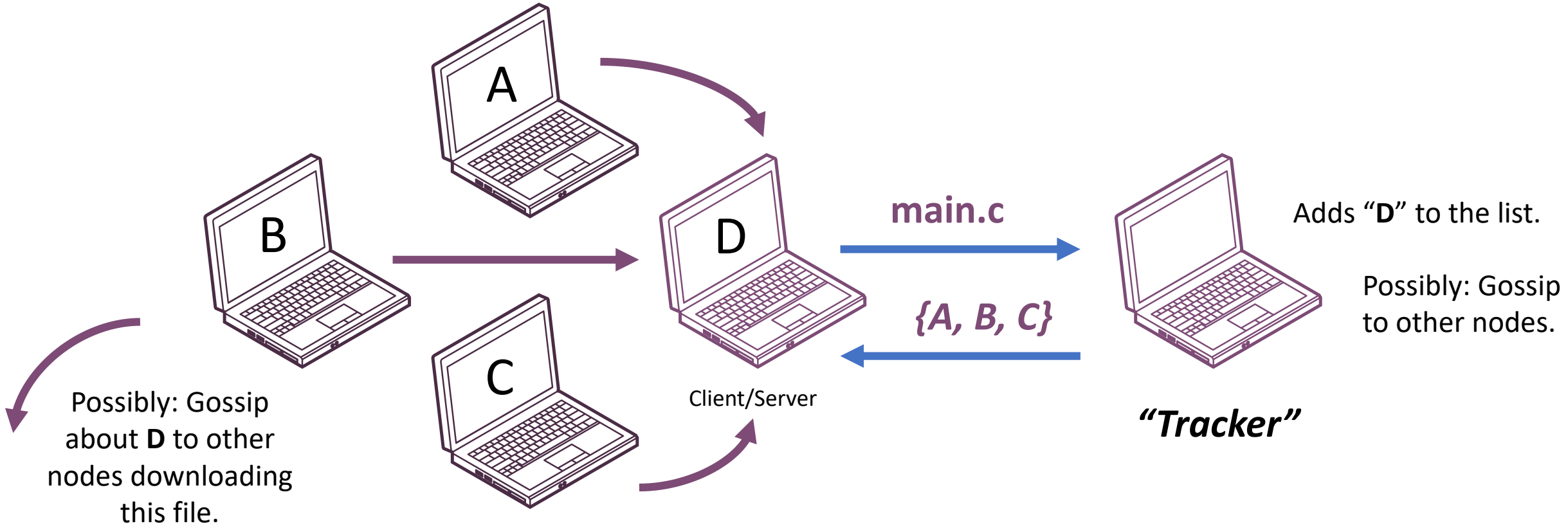
- A basic peer-to-peer system based on block swapping.
- These days built on top of Distributed Hash Tables (DHTs)
- Known in non-technical circles for its use within software piracy.
  - But it, or something similar, is used often!
  - Blizzard has game download and WoW updates happen via BitTorrent.
  - Many Linux distributions allow downloading them via BitTorrent.

  - AT&T said in 2015 that BitTorrent represented around 20% of total broadband bandwidth: https://thestack.com/world/2015/02/19/att-patents-system-to-fast-lane-bittorrent-traffic/
    - I'm actually a bit skeptical.

BitTorrent System Model

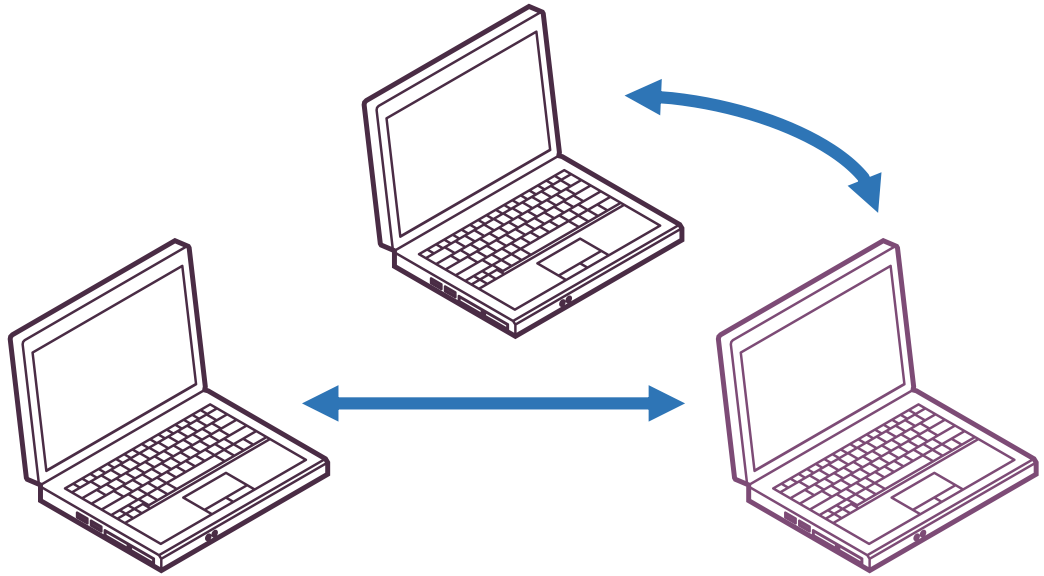
When a file is requested, a well-known node yields a peer list.

Our node serves as both client ***and*** server. (As opposed to unidirectional NFS)



BitTorrent Block Sharing

- Files are divided into chunks (blocks) and traded among the different peers.
- As your local machine gathers blocks, those are available for other peers, who will ask you for them.



Client/Server

- You can concurrently download parts of files from different sources.
- Peers can leave and join this network at any time.

Heuristics for Fairness

- How to choose who gets a block? (No right/obvious answer)
  - This is two-sided. How can you trust a server to give you the right thing?
  - Some peers are faster/slower than others.
  - In an open system: Some don't play fair. They take but never give back.
- You could prioritize older nodes.
  - They are less likely to suddenly disappear.
  - They are more likely to cooperate.
  - What if everybody did this… hmm… old nodes shunning young nodes…

(The Millennial Struggle, am I right?)

- You can only give if the other node gives you a block you need.
  - Fair Block/Bit-swapping. Works as long as you *have some data*.
  - Obviously punishes first-timers (who don't have any data to give)
  - Incentivizes longevity with respect to cooperation.

Centralization Problem

- “Tracker” based solution introduces unreliable ***centralization***.

- Getting rid of that (decentralized tracking) means:
  - Organizing nodes such that it is easy to find data.
  - Yet, also, not requiring knowledge about where that data is.
  - And therefore, allowing data to move (migrate) as it sees fit.

- Many possible solutions. Most are VERY interesting and some are slightly counter-intuitive (hence interesting!)

Distributed Hash Tables (DHT)

- A distributed system devoted to decentralized key/value storage across a (presumably large or global) network.

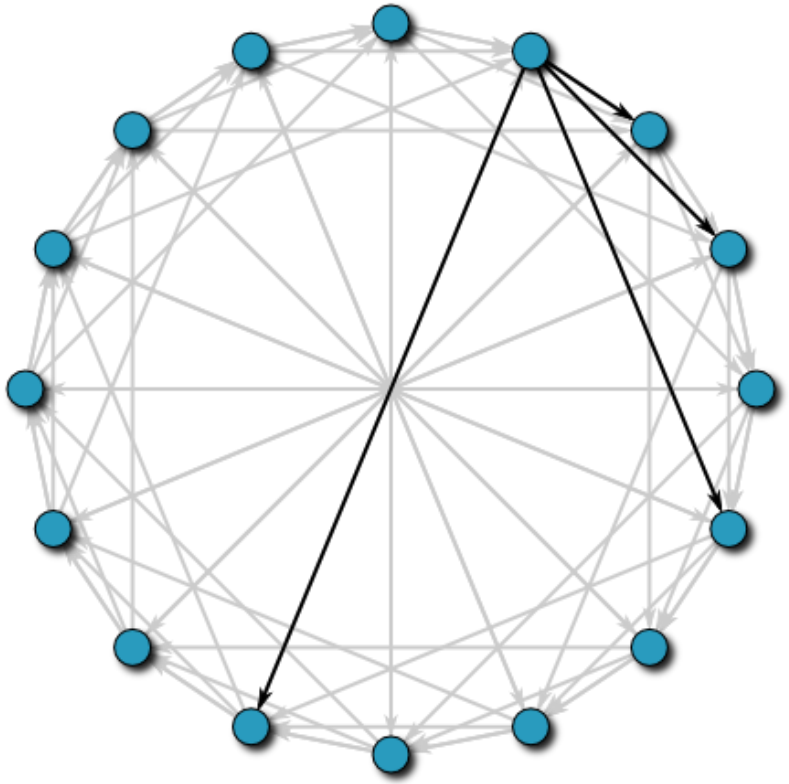
- These are “tracker”-less. They are built to not require a centralized database matching files against peers who have them.

- Early DHTs were motivated by peer-to-peer networks.
  - Early systems (around 2001): Chord, Pastry, Tapestry
  - All building off one another.

Distributed Hash Tables: Basics

- Files are content-addressed and stored by their hash (key).
- Fulfills one simple function: $\text{value} = lookup(key)$
- However, the value could be anywhere! IN THE WORLD. Hmm.
- Many find a way to relate the key to the location of the server that holds the value.
- The goal is at $O(\log N)$ queries to find data.
  - Size of your network can increase exponentially as lookup cost increases linearly. (Good if you want to scale to millions of nodes)

Chord DHT

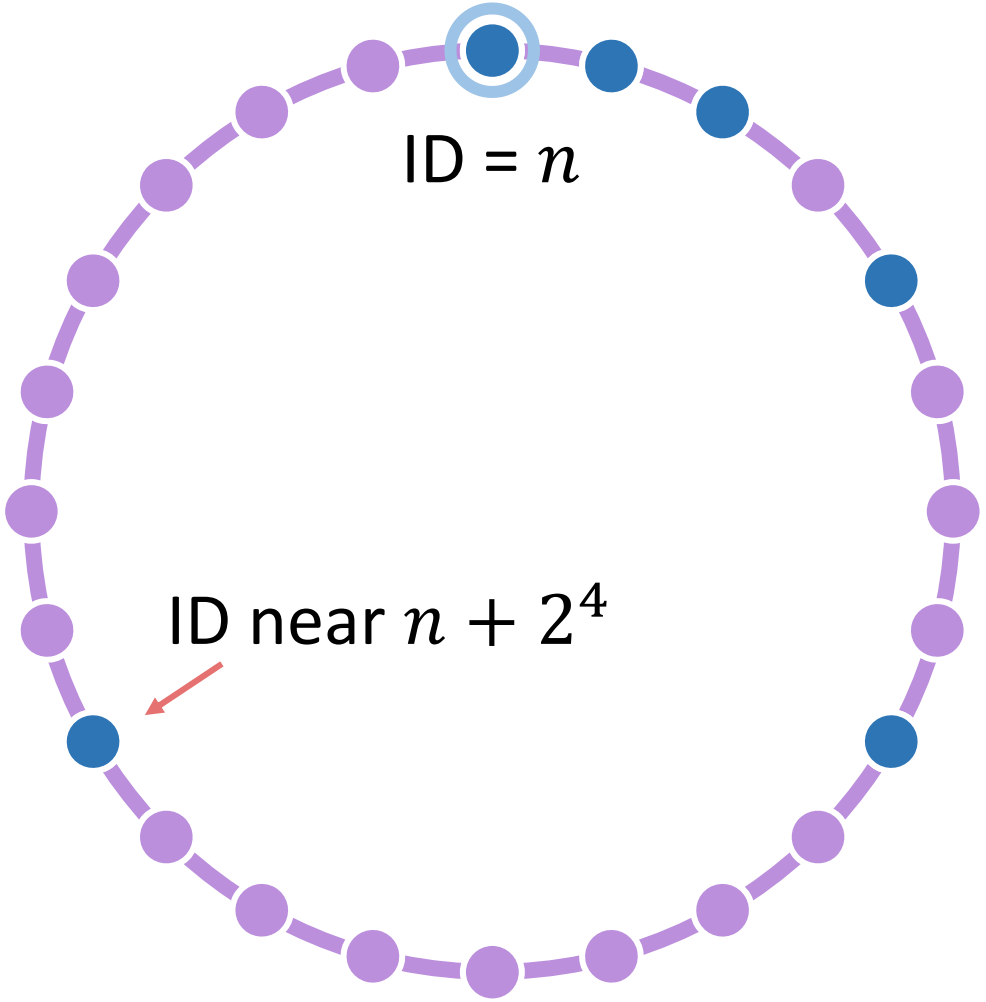


16 Node Network

(image via Wikipedia)

- Peers are given an ID as a hash of their IP address. (unique, uniform)
- Such nodes maintain information about files that have hashes that resemble their IDs. (Distance can be the difference: A-B)
- Nodes also store information about neighbors of successive distances. (very near, near, far, very far… etc)
- Organizes metadata across the network to reduce the problem to a binary search.
  - Therefore needs to contact $O(\log N)$ servers.
- To find a file, contact the server with an ID equal or slightly less than the file hash.
  - They will then reroute to their neighbors. Repeat.

Chord System Model

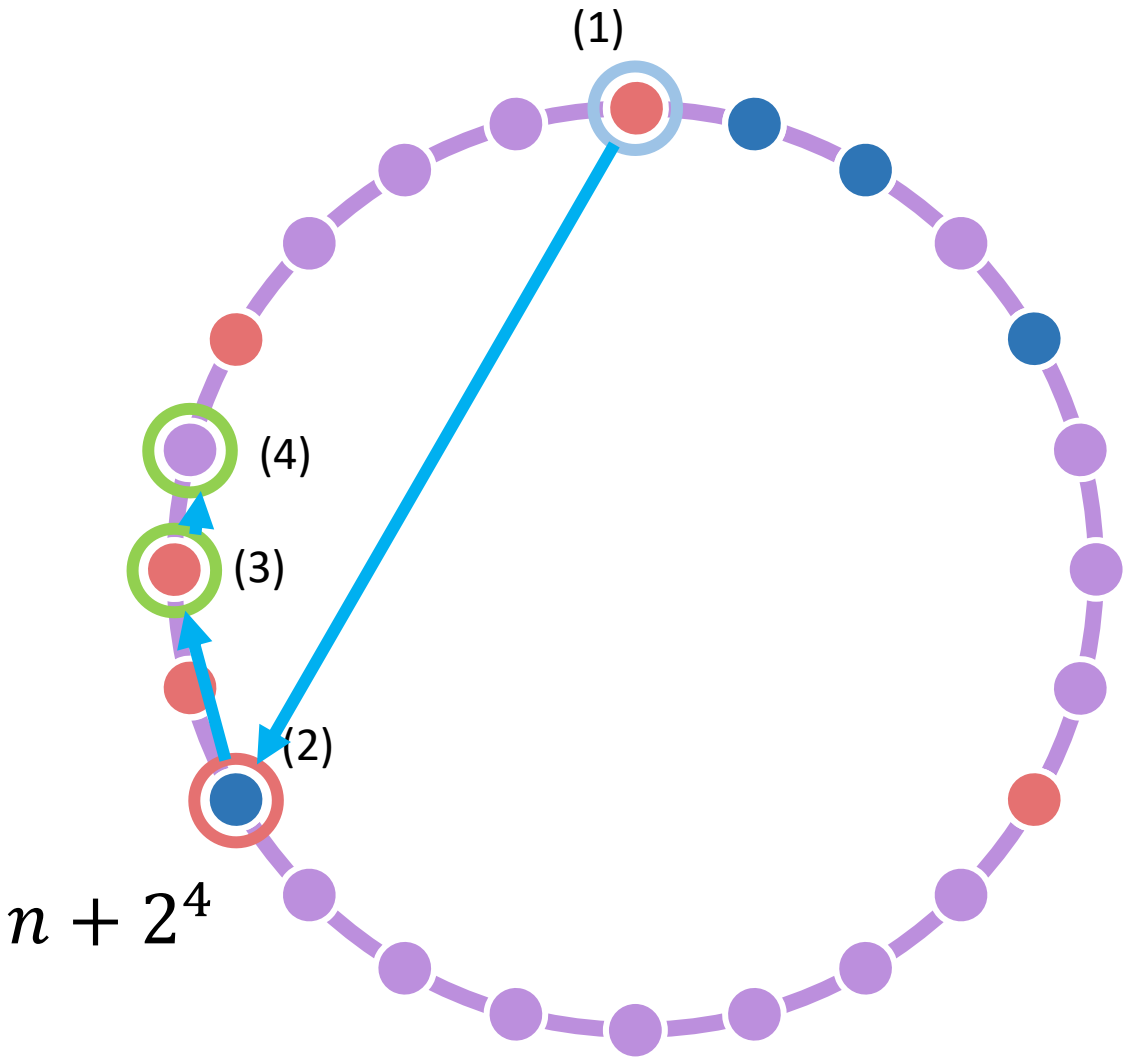


- Nodes are logically organized into a ring formation sorted by their ID (n).
  - IDs increase as one moves clockwise.
  - IDs should have the same bit-width as the keys.
  - For our purposes, keys are file hashes.

- Nodes store information about neighbors with IDs relative to their own in the form: (m is key size in bits)
  - $\left(n + 2^i\right) \bmod 2^m$ where $0 \leq\ i < m$

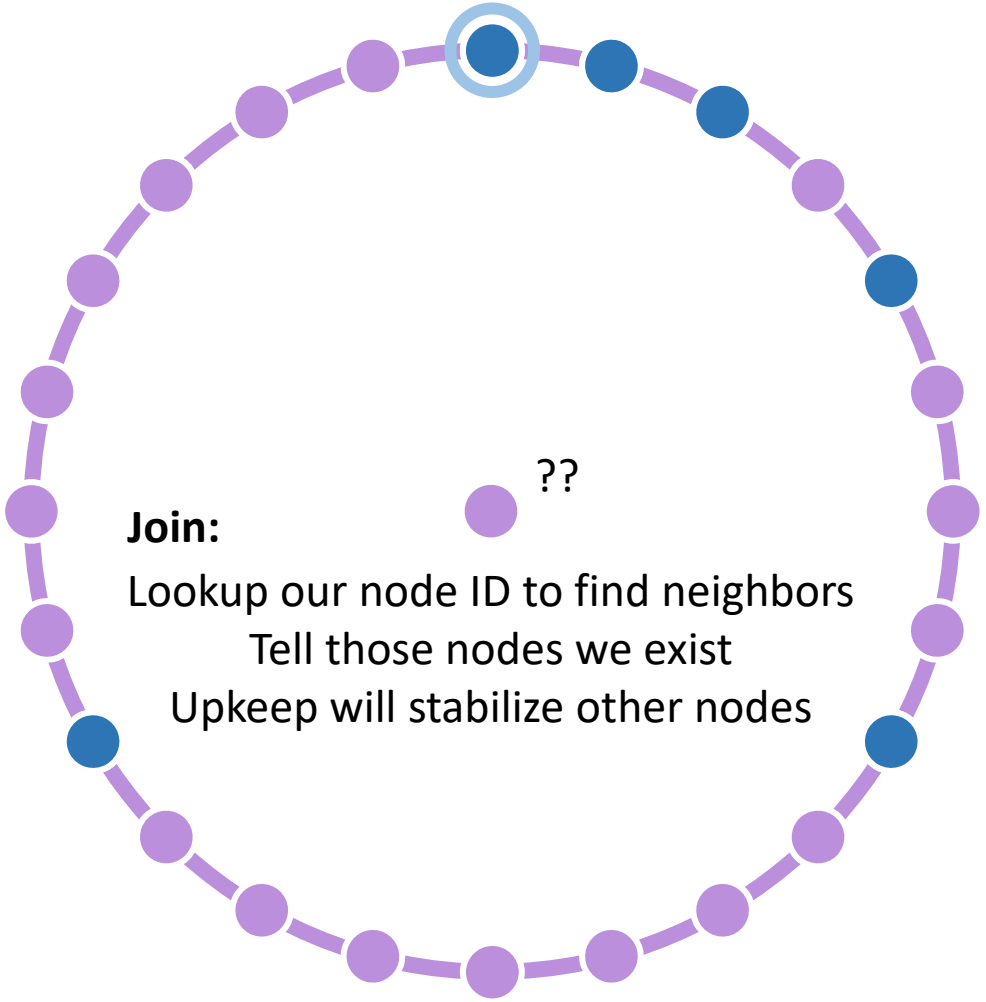
- Imagine a ring with *millions* of nodes.
  - 2^i diverges quickly!

Chord: Lookup



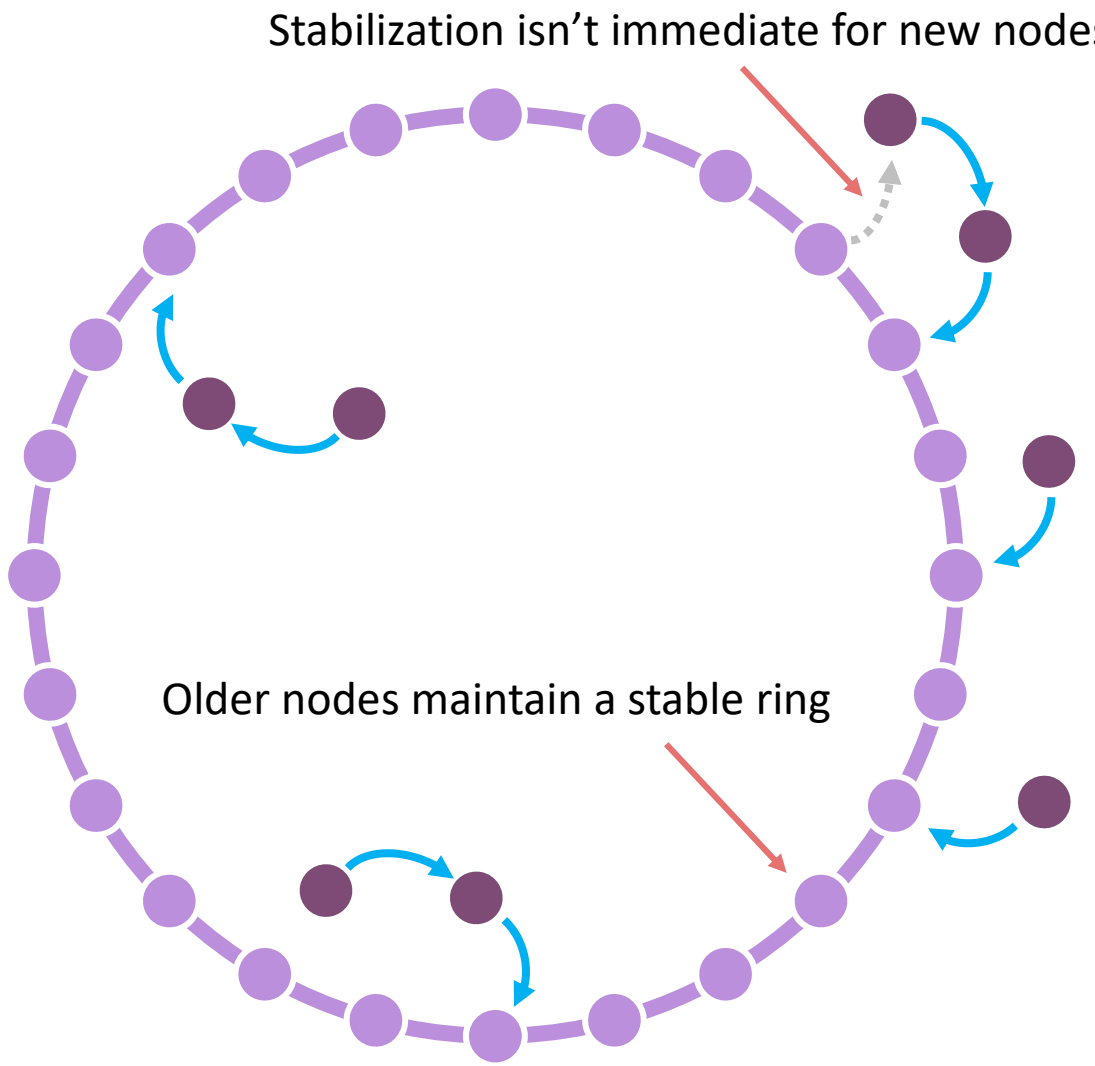
- Notice how locality is encoded.
  - Nodes know at most $\log m$ nodes.
  - Nodes know more “nearby” nodes.
- When performing $lookup(key)$, the node only needs to find the node closest to that key and forward the request.
- Let’s say key is far away from us.
  - We will ask the node farthest from us (with the “nearest” ID less than the key)
- This node, as before, also knows about neighbors in a similar fashion.
  - Notice it’s own locality! It looks up the same key. Binary search… $O(\log N)$ msgs.

Chord: Upkeep, Join



- Periodically, the node must check to ensure it's perception of the world (the ring structure) is accurate.
- It can ask its neighbor who their neighbor is.
  - If it reports a node whose ID is closer to $n + 2^i$ than they are… use them as that neighbor instead.
- This is done when a node enters the system as well.
  - All new neighbors receive information about, and responsibility for, nearby keys.

Problems with Chord

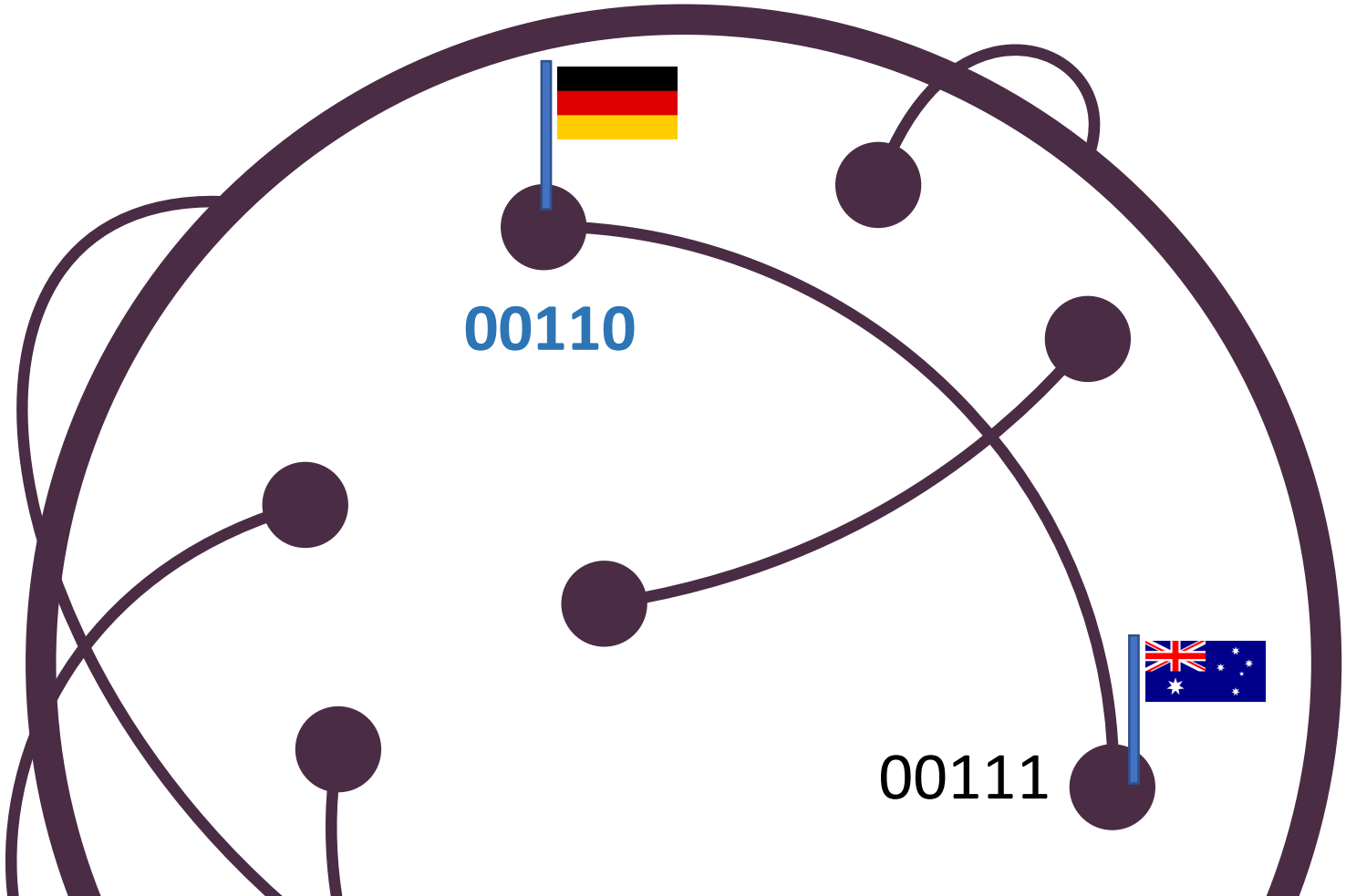


- Maintaining the invariants of the distributed data structure is hard.
  - That is, the ring shape.
- When new nodes enter, they dangle off of the ring until nodes see them.
- That means, it doesn't handle short-lived nodes very well.
  - Which can be very common for systems with millions of nodes!

Kademlia (Pseudo Geography)

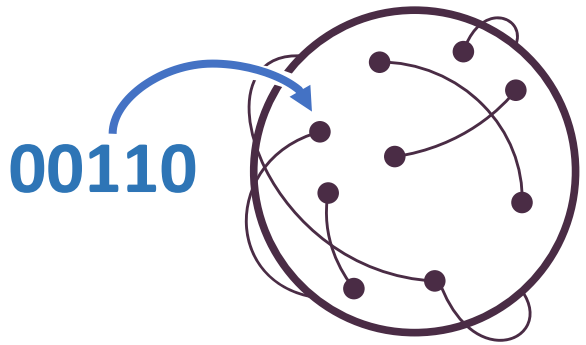
- Randomly assign yourself a node ID ☺
- Measure distance using XOR: $d(N_1, N_2) = N_1 \oplus N_2$ (Interesting…)
  - Unlike arithmetic difference (A – B) no two nodes can have the same distance to any key.
  - XOR has the same properties as Euclidian distance, but cheaper:
    - Identity: $d(N_1, N_1) = N_1 \oplus N_1 = 0$
    - Symmetry: $d(N_1, N_2) = d(N_2, N_1) = N_1 \oplus N_2 = N_2 \oplus N_1$
    - Triangle Inequality: $d(N_1, N_2) \leq d(N_1, N_3) + d(N_2, N_3)$
      $N_1 \oplus N_2 \leq (N_1 \oplus N_3) + (N_2 \oplus N_3)$ … Confounding, but true.
- Once again, we store keys near similar IDs.
  - This time, we minimize the distance:
    - Store key k at any node n that minimizes $d(n, k)$

Kademlia Network Topology



- Two “neighbors” may be entirely across the planet! (or right next door)

Kademlia Network Topology



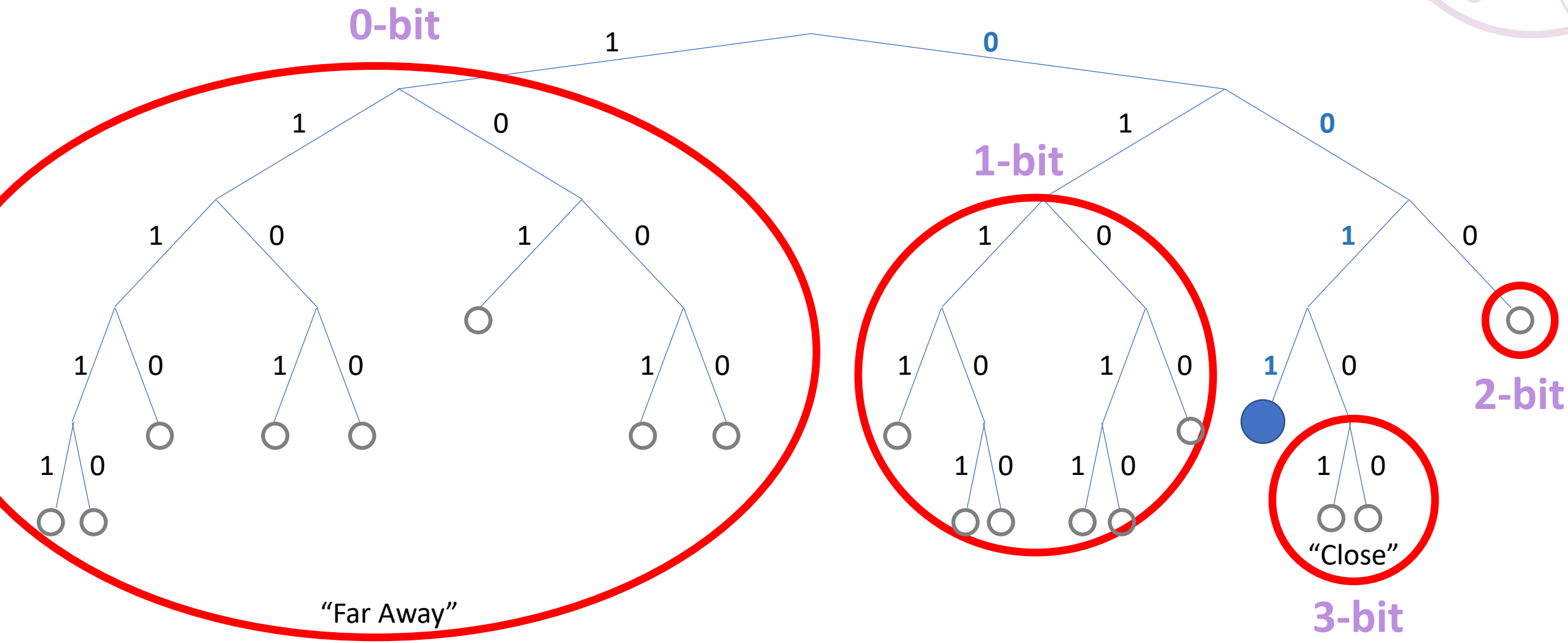
Routing Table k-buckets

0-bit	1-bit	2-bit	3-bit	4-bit
10001	01001	00011	00100	00111
10100	01100	00010	00101	
10110	01010	00001		
11001	01001	00000		

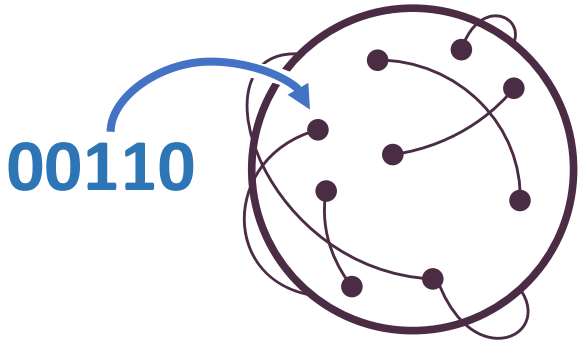
Note: 0-bit list contains half of the overall network!

- Each node knows about nodes that have a distance successively larger than it.
  - Recall XOR is distance, so largest distance occurs **when MSB is different**.
- It maintains buckets of nodes with IDs that share a **prefix** of k bits (matching MSBs)
  - There are a certain number of entries in each bucket. (not exhaustive)
  - The number of entries relates to the replication amount.
- The overall network is a **trie**.
  - The buckets are *subtrees* of that trie.

Kademlia Routing (bucket visualization)



Kademlia Routing Algorithm



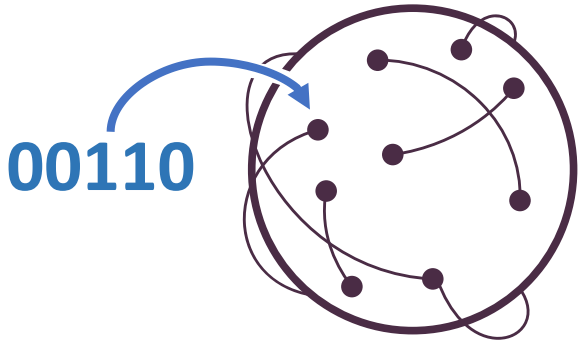
Routing Table k-buckets

0-bit	1-bit	2-bit	3-bit	4-bit
10001	01001	00011	00100	00111
10100	01100	00010	00101	
10110	01010	00001		
11001	01001	00000		

Note: 0-bit list contains half of the overall network!

- Ask the nodes we know that are "close" to k to tell as about nodes that are "close" to k
- Repeat by asking *those* nodes which nodes are "close" to k until we get a set that say "I know k!!"
- Because of our k-bucket scheme, each step we will look at nodes that share an increasing number of bits with k.
  - *And because of our binary tree, we essentially divide our search space in half.*
  - Search: $O(\log N)$ queries.

Kademlia Routing Algorithm



Routing Table k-buckets

0-bit	1-bit	2-bit	3-bit	4-bit
10001	01001	00011	00100	00111
10100	01100	00010	00101	
10110	01010	00001		
11001	01001	00000		

Note: 0-bit list contains half of the overall network!

- Finding $k = 00111$ from node 00110.
  - Easy! Starts with a similar sequence.
  - It's hopefully at our own node, node 00111, or maybe node 00100…
- Finding $k = 11011$ from 00110:
  - Worst case! No matching prefix!
  - Ask *several* nodes with IDs starting with 1.
    - This is, at worst, half of our network… so we have to rely on the algorithm to narrow it down.
    - It hopefully returns nodes that start with 11 or better. (which eliminates another half of our network from consideration)
  - Repeat until a node knows about k.

Kademlia: Node Introduction

- Contrary to Chord, XOR distance means nodes know exactly where they fit.
  - How “far away” you are from any key doesn’t depend on the other nodes in the system. (It’s always your ID $\oplus$ key)
- Regardless the join process is more or less the same:
  - Ask an existing node to find your ID, it returns a list of your neighbors.
  - Tell your neighbors you exist and get their knowledge of the world
    - That is, replicate their keys and k-buckets.
- As nodes contact you, record their ID in the appropriate bucket.
  - When do you replace?? Which entries do you replace?? Hmm.

Applications

- IPFS (InterPlanetary File System)
  - Divides files into hashes resembling a Merkle DAG.
  - Uses a variant of Kademlia to look up each hash and find mirrors.
  - Reconstructs files on the client-side by downloading from peers.
  - Some very shaky stuff about using a blockchain (distributed ledger) to do name resolution.
  - Is this the next big thing??? (probably not, but it is cool ☺)