P2P

SWE 622, Spring 2017
Distributed Software Engineering

GEORGE MASON UNIVERSITY

HW4

- Grading is mostly done (sorry!)
- Main issue: handling ZK disconnections
  - Shouldn't allow R/W operations
  - GroupMember will report a cached view of currentMembers - so if you are disconnected it’s likely to be wrong (or if you never connected)
  - lock.acquire()/release() can throw a ConnectionLostException

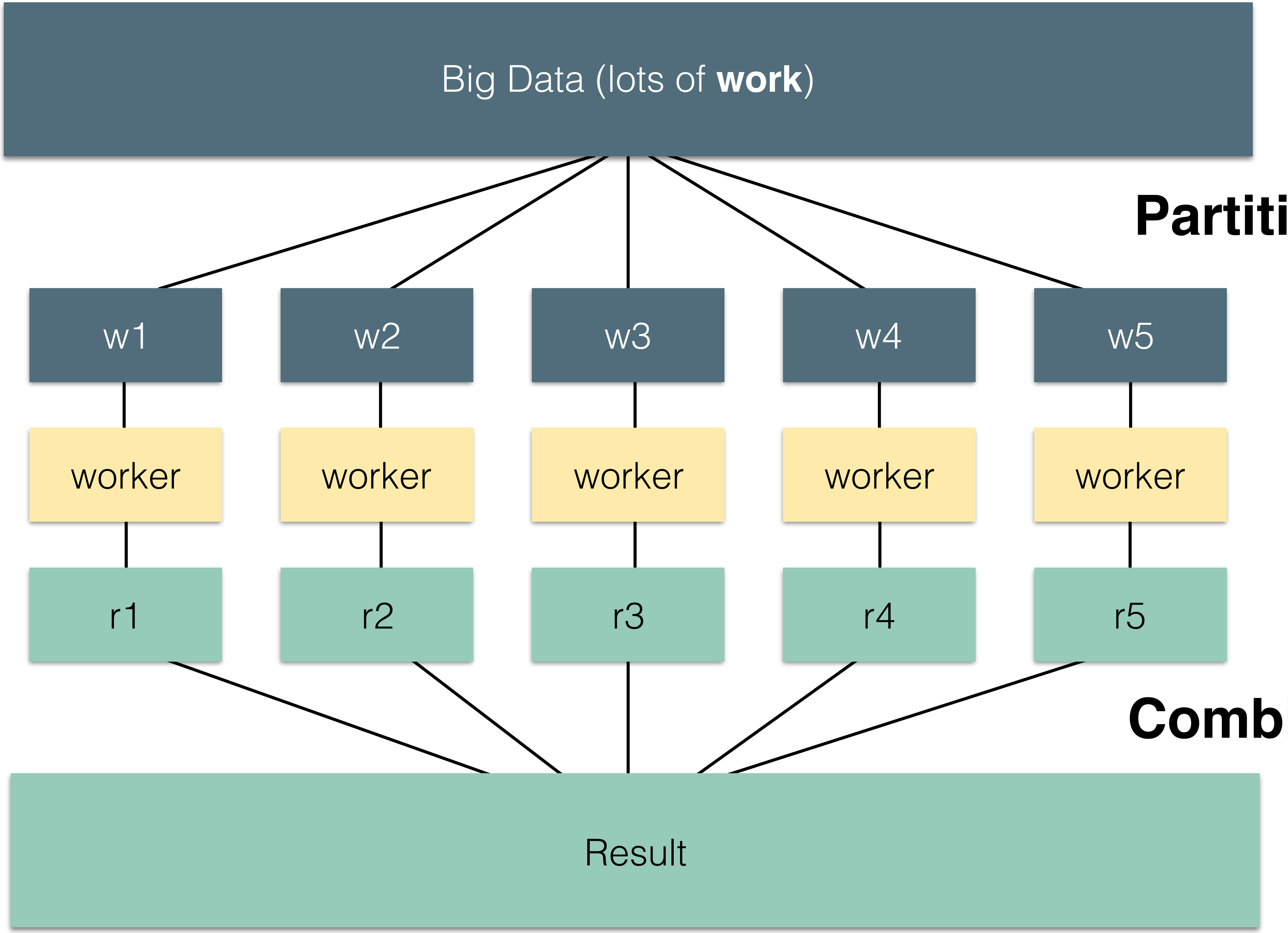
HW4: Handling Locking Errors

- OK: Re-throw exceptions thrown by ZK
- Not OK: try{lock()} catch(Exception e) { println("Error!"); }
- OK: Using timeouts
- Not OK: Using timeouts but not throwing an error if unable to proceed

Review: Distributing Computation

- Lots of these challenges re-appear, regardless of our specific problem
  - How to split up the task
  - How to put the results back together
  - How to store the data
- Enter, MapReduce

MapReduce: Divide & Conquer



Big Data (lots of **work**)

Partition

w1 w2 w3 w4 w5

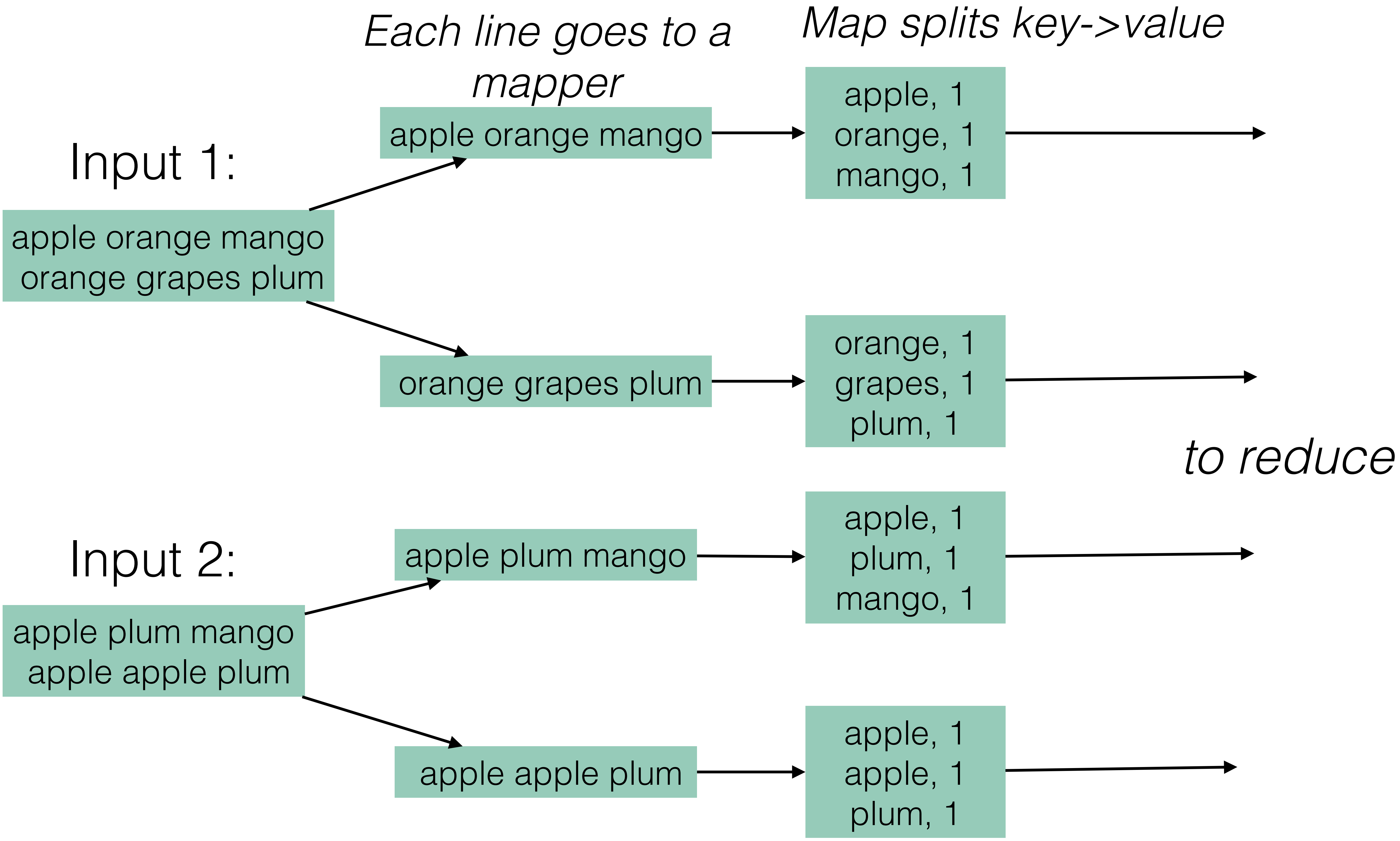
worker worker worker worker worker

r1 r2 r3 r4 r5

Combine

Result

MapReduce: Example



Each line goes to a mapper

Map splits key->value

Input 1:

apple orange mango
orange grapes plum

apple orange mango

apple, 1
orange, 1
mango, 1

orange grapes plum

orange, 1
grapes, 1
plum, 1

to reduce

Input 2:

apple plum mango
apple apple plum

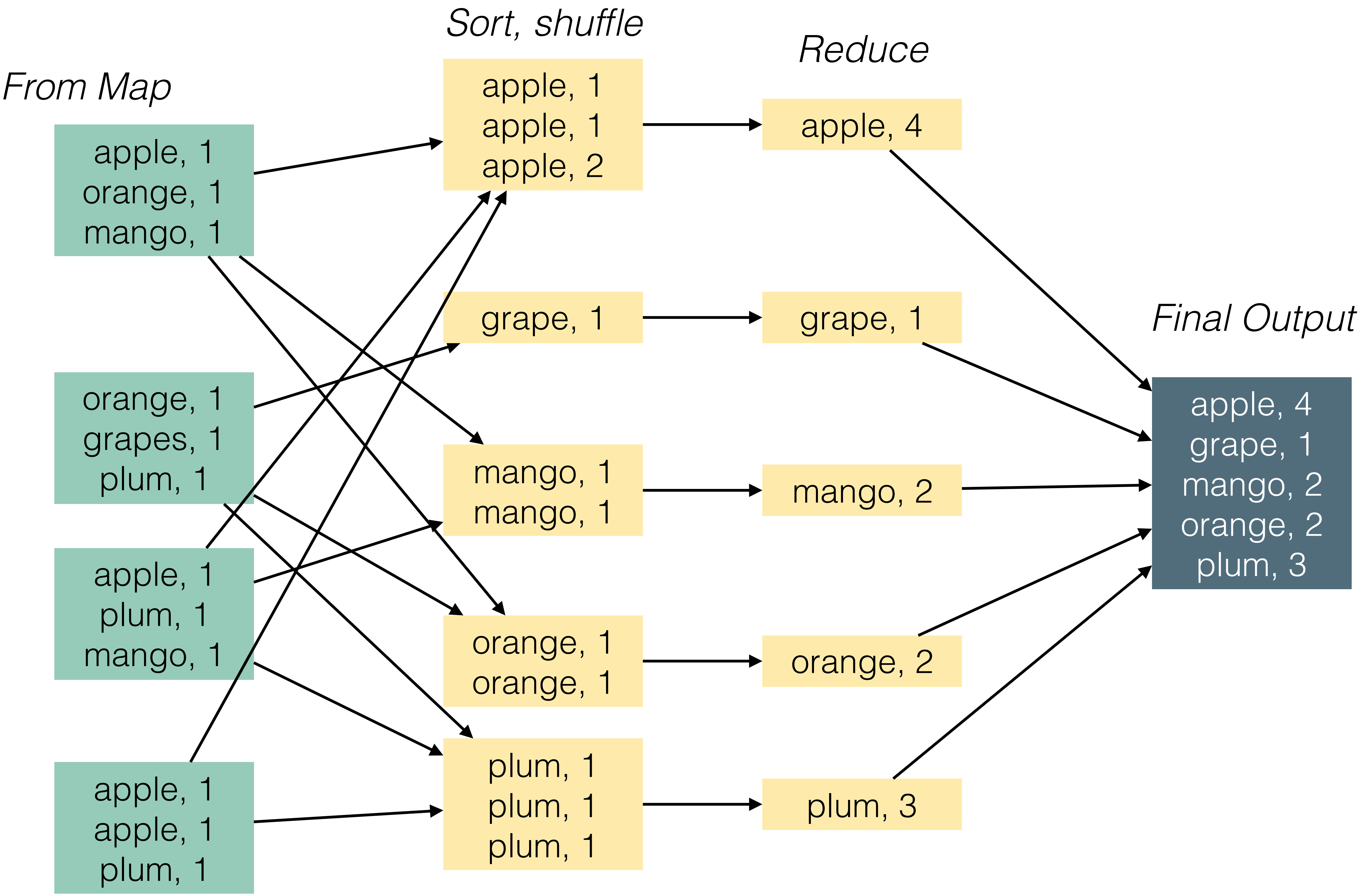
apple plum mango

apple, 1
plum, 1
mango, 1

apple apple plum

apple, 1
apple, 1
plum, 1

MapReduce: Example



From Map

apple, 1
orange, 1
mango, 1

orange, 1
grapes, 1
plum, 1

apple, 1
plum, 1
mango, 1

apple, 1
apple, 1
plum, 1

Sort, shuffle

apple, 1
apple, 1
apple, 2

grape, 1

mango, 1
mango, 1

orange, 1
orange, 1

plum, 1
plum, 1
plum, 1

Reduce

apple, 4

grape, 1

mango, 2

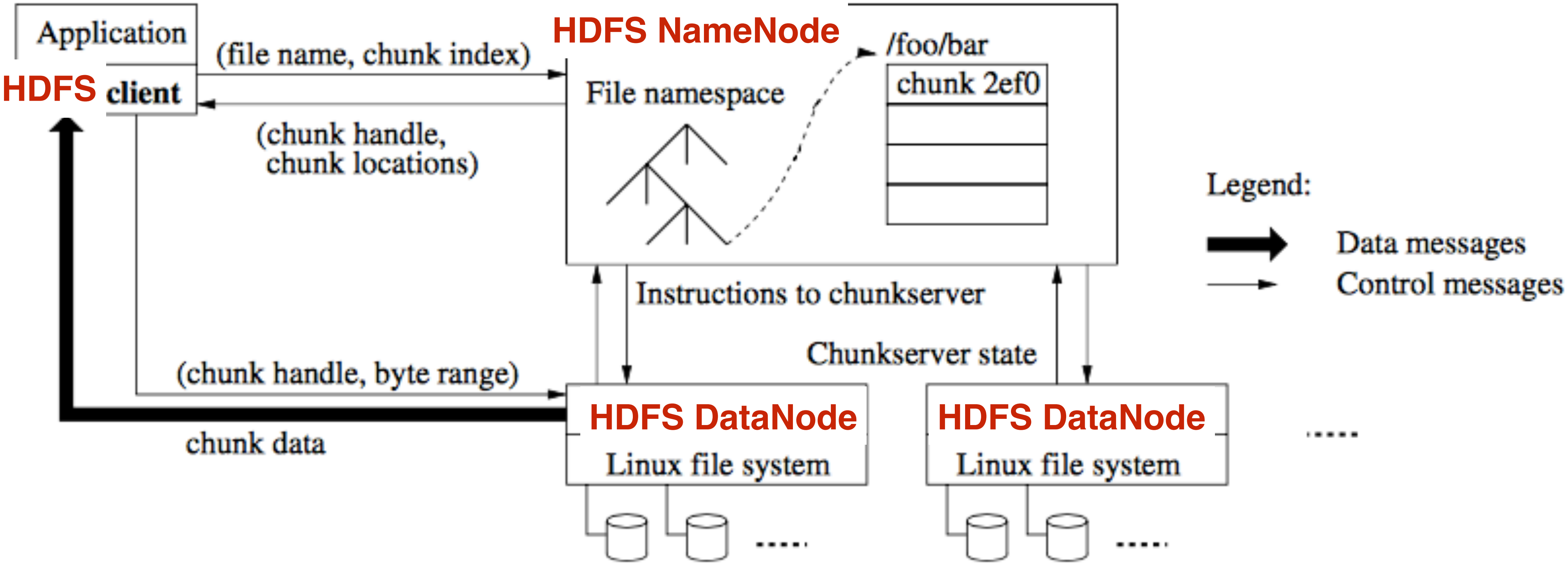
orange, 2

plum, 3

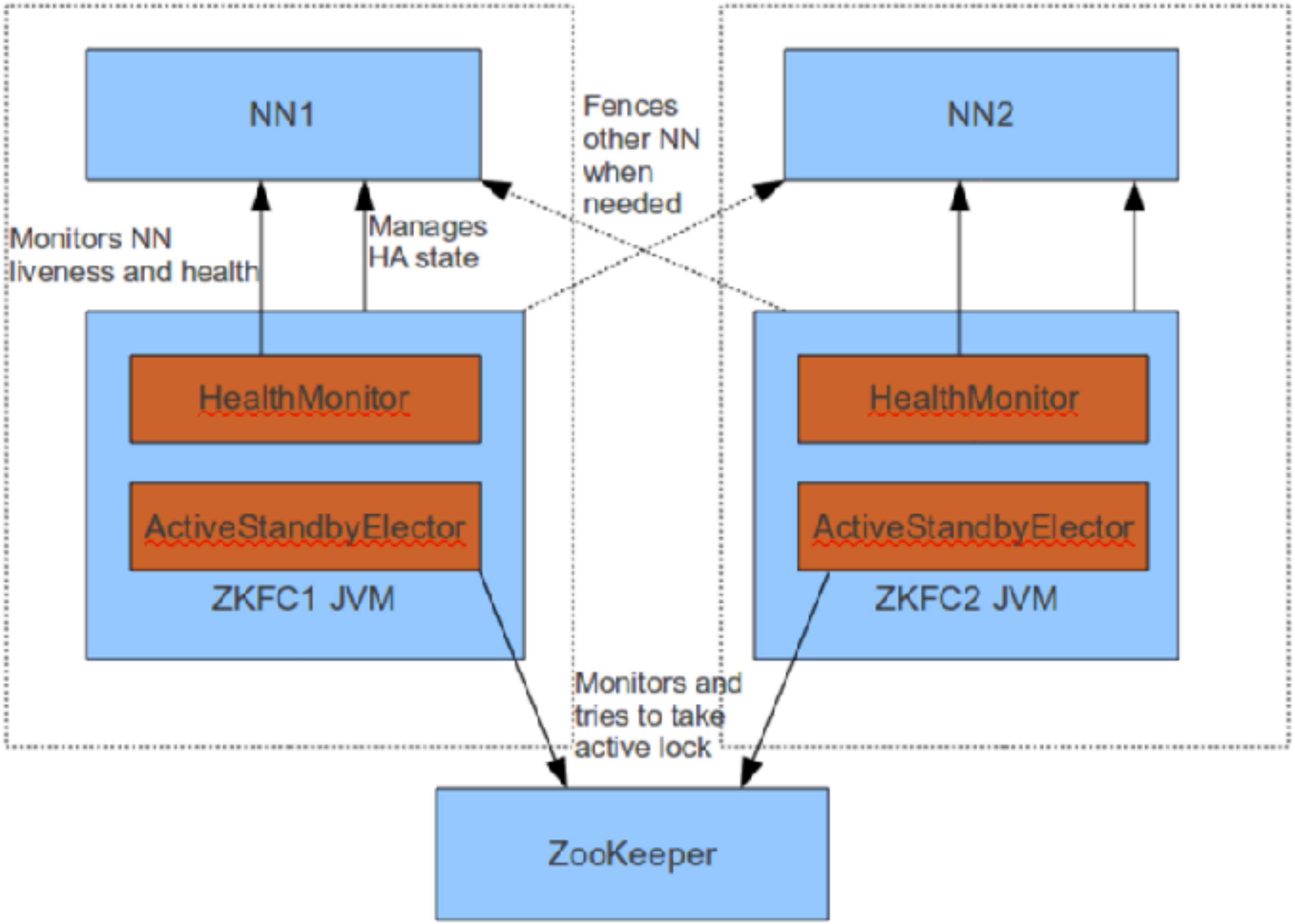
Final Output

apple, 4
grape, 1
mango, 2
orange, 2
plum, 3

Hadoop: HDFS

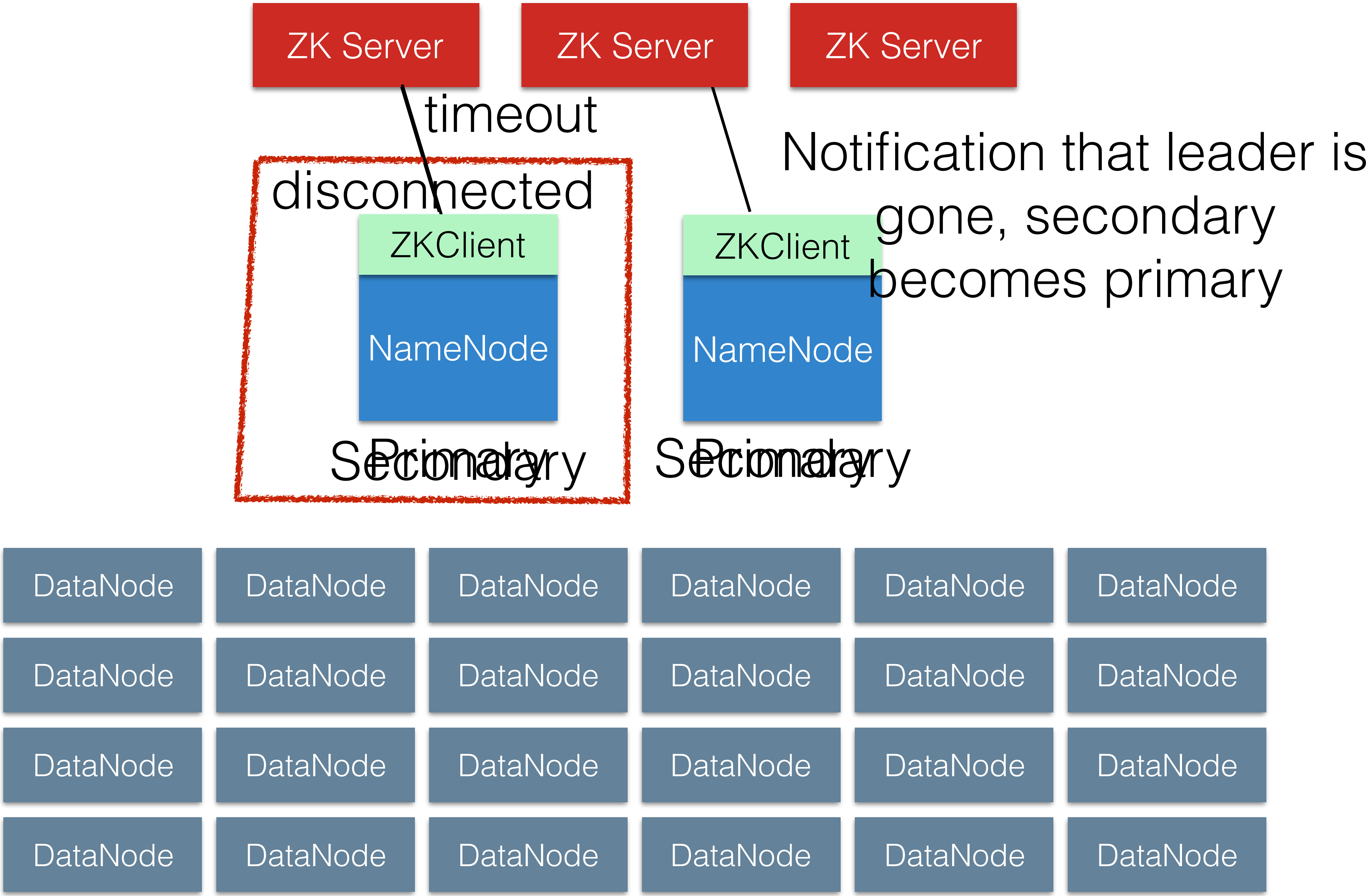

Application
HDFS client
(file name, chunk index)
(chunk handle, chunk locations)
HDFS NameNode
File namespace
/foo/bar
chunk 2ef0
Legend:
Data messages
Control messages
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
HDFS DataNode
Linux file system
HDFS DataNode
Linux file system

Hadoop + ZooKeeper



https://issues.apache.org/jira/secure/attachment/12519914/zkfc-design.pdf

Hadoop + ZooKeeper



Notification that leader is gone, secondary becomes primary

Spark

- Aggressively caches data in memory
- *AND* is fault tolerant
- How? MapReduce got tolerance through its disk replication
- RDDs are *resilient* but they are also **restricted**
  - Immutable, partitioned records
  - Can only be built through coarse-grained and **determistic** transformations (e.g. map, filter, join)

Role of ZK in CFS for fault tolerance

- We will track who the Redis master is with ZK
- If something is wrong (WAIT fails) or can't contact master, then you need to challenge the master's leadership
  - Maybe master is partitioned from slaves but ZK isn't
  - Maybe master is crashed
- If you promote a master, then you need to track that in ZK
- If you were disconnected and reconnect from ZK, you need to validate who the master is

HW 5: Fault Tolerance

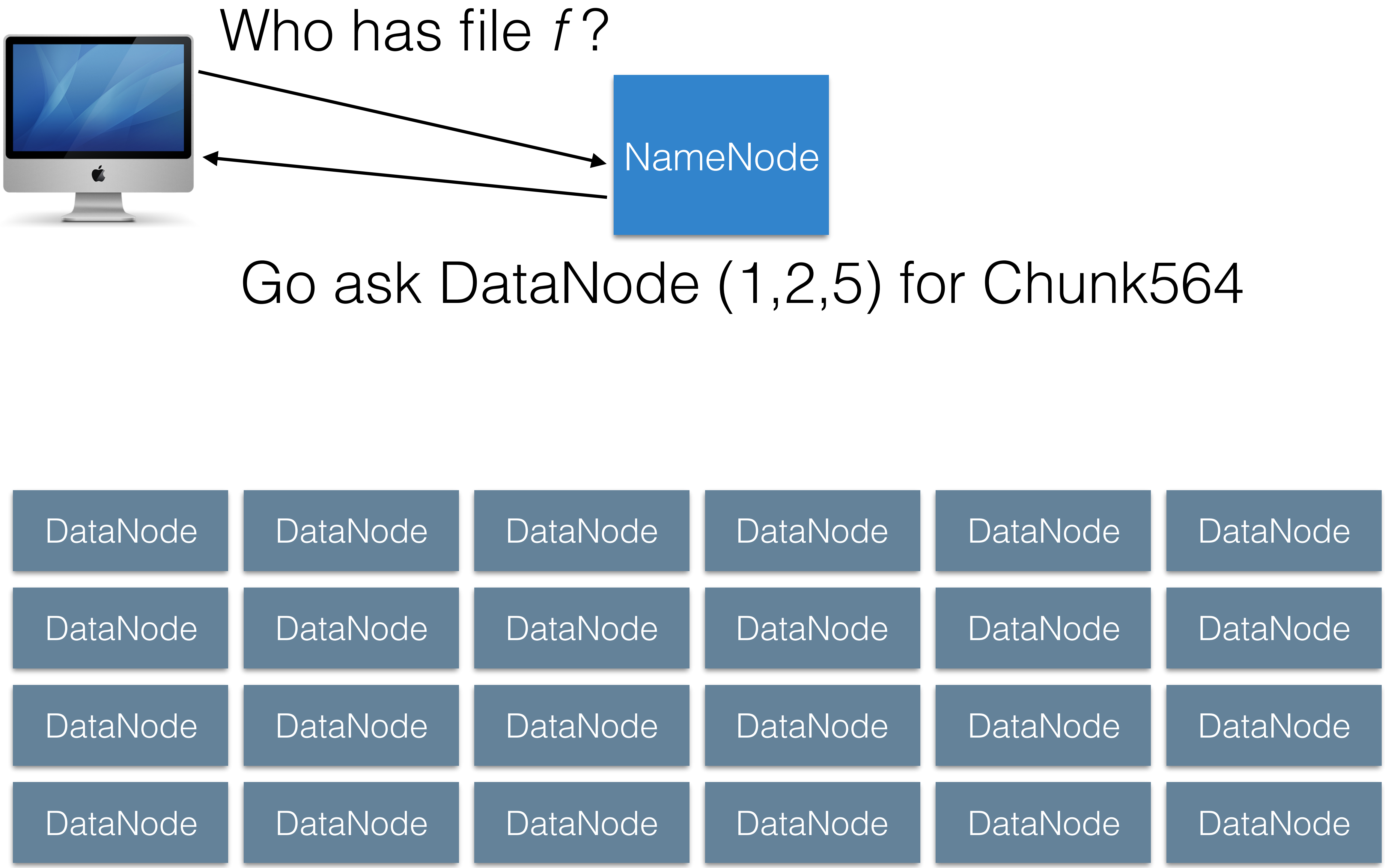
- We're going all-in on ZooKeeper here
- Use ZK similar to how HDFS does: each Redis slave will have a dedicated ZK client to determine who the master Redis server is
- Redis master holds a lease that can be renewed perpetually
- When client notices a problem (e.g. WAIT doesn't work right or can't talk to master) it proposes becoming the master
- As long as a client can talk to a quorum of ZK nodes, then they can decide who the leader is
- Clients don't need to vote - just matters that there is exactly one of them

Today

- Finding where to find data
- P2P (Napster, Gnutella, BitTorrent)
- CDNs
- Consistency tradeoffs?

Where do we find data?

The easy answer: master metadata server (GFS, HDFS, etc)



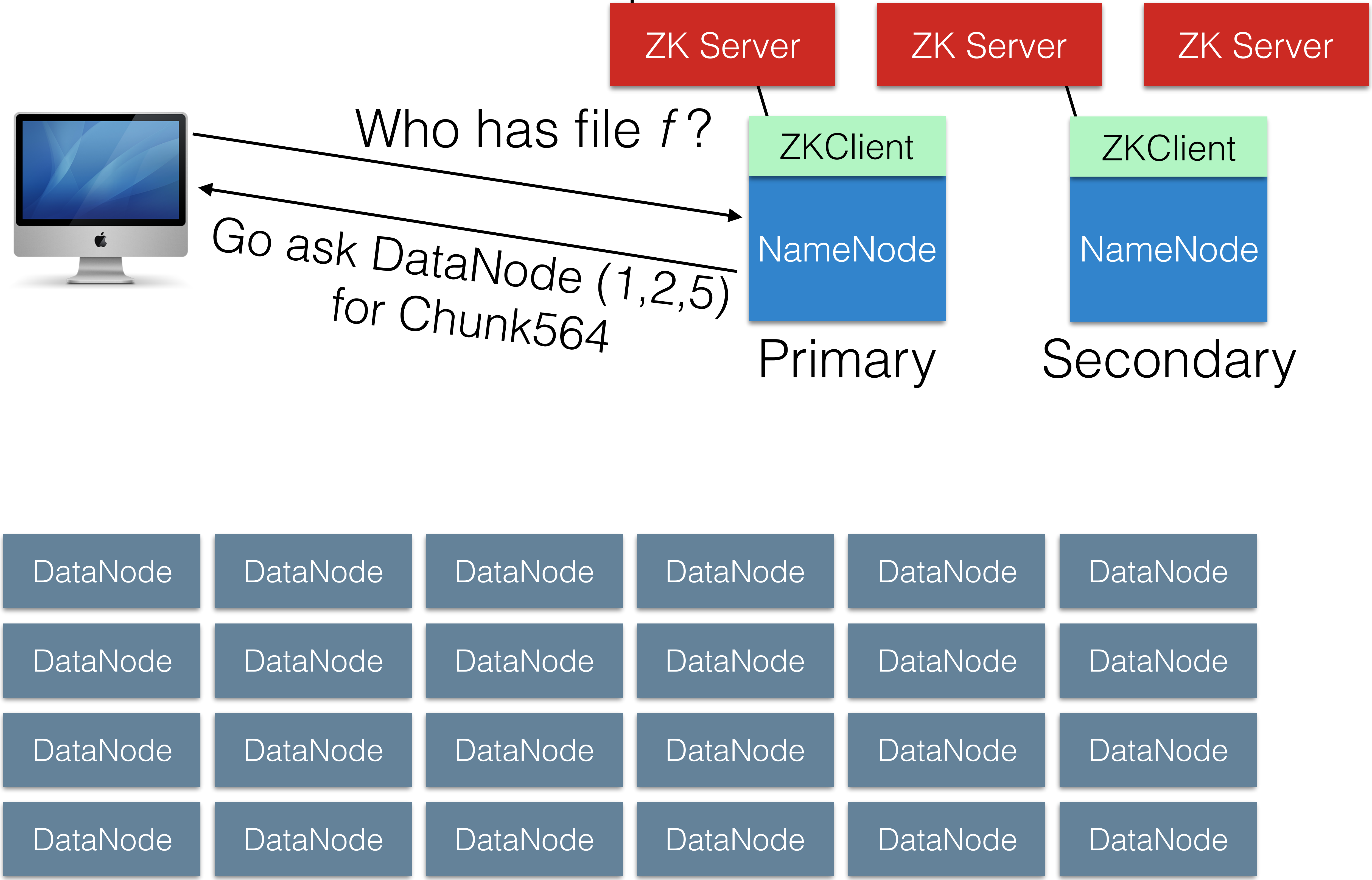
Who has file *f* ?

NameNode

Go ask DataNode (1,2,5) for Chunk564

Where do we find data?

The fault tolerant answer: replicate that metadata server



ZK Server

ZK Server

ZK Server

Who has file *f* ?

Go ask DataNode (1,2,5) for Chunk564

ZKClient

NameNode

Primary

ZKClient

NameNode

Secondary

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

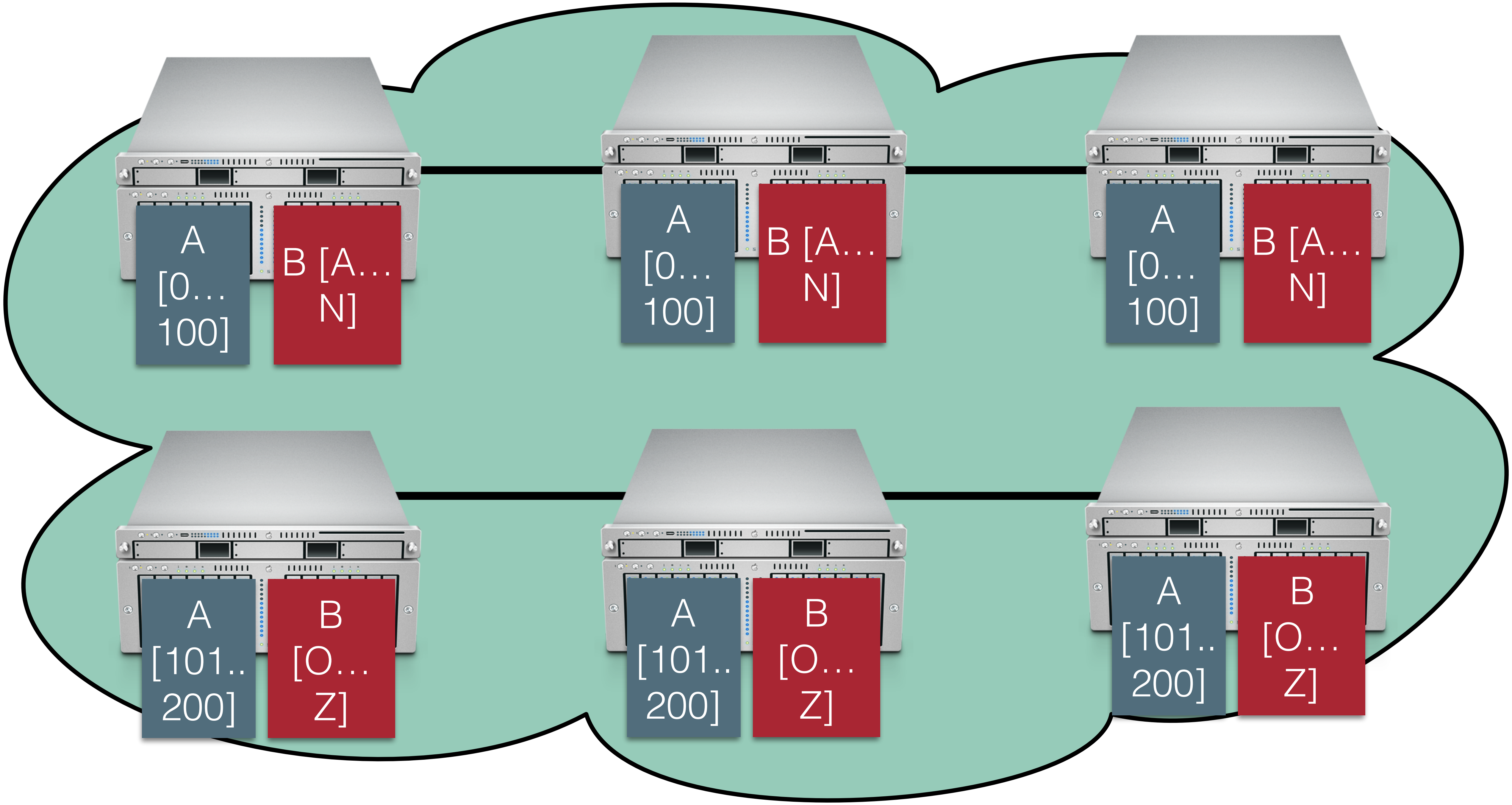
Example

- Let's build a system to track how many times each user views our web page
- I have hundreds of millions of users
- Needs to be fast, available

Where do we find data?

- What's bad with the single master picture?
- HDFS/GFS leverage the fact that there is relatively **little** metadata, **lots** of data (e.g. few files, each file is large)
- What if there is really only metadata?
- How can we build a system with high performance **without** having this single server that knows where data is stored?

Partitioning + Replication



Partitioning + Replication

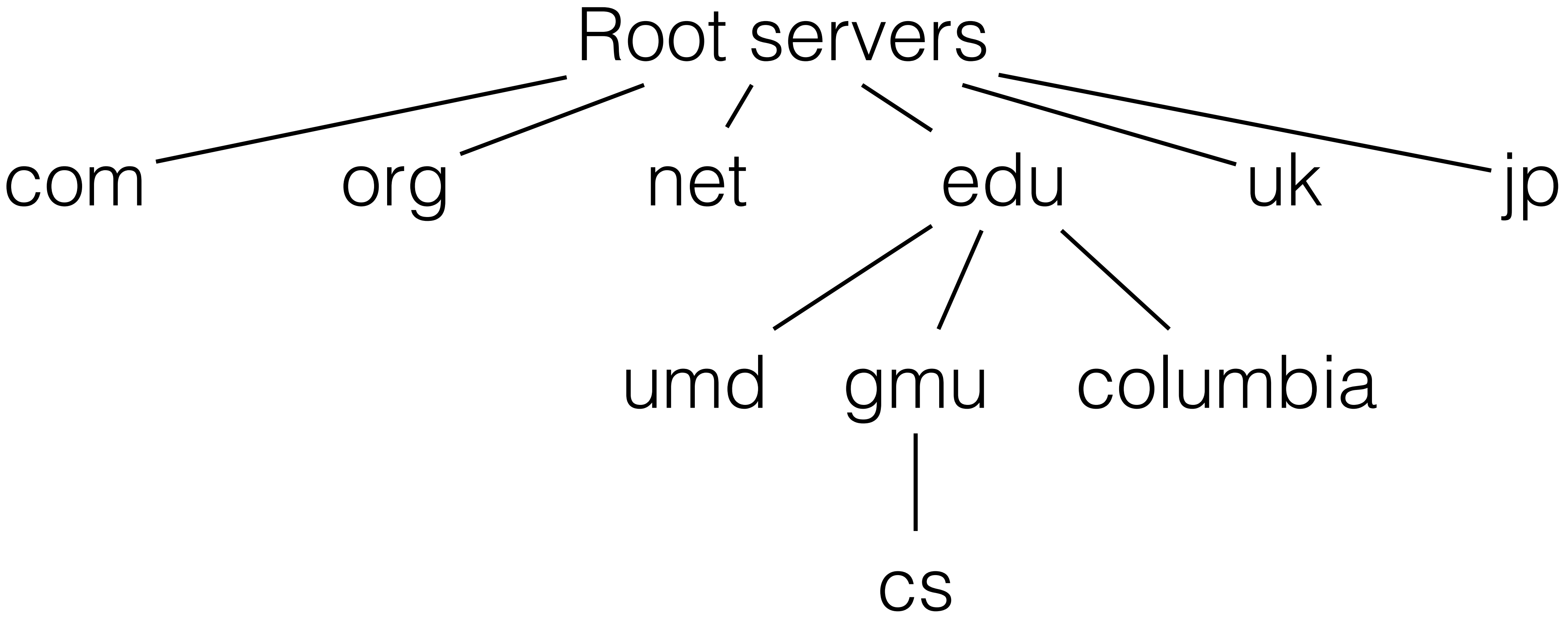
- We can solve our **discovery** problem if we can define a **consistent** way to store our data and share those rules
- Create "buckets," and use a "hash function" to map from a key to a bucket
- Example: All files starting with the letter "A" are stored on servers 1,2,3; all files starting with the letter "B" are stored on severs 4,5,6, etc.
- Problems with this approach:
  - What if files are not evenly distributed among buckets?

Partitioning + Replication

- If input is structured, can possibly leverage that structure to build these buckets (**name spaces**)
- Example: File system
  - Map from: /home/bellj/teaching/swe622 to file contents
  - Could have different machines responsible for each tier?
- Example: DNS system
  - Maps from: www.jonbell.net TO: 104.24.122.171
  - Different machines for each tier?

DNS

Idea: break apart responsibility for each part of a domain name (**zone**) to a different group of servers

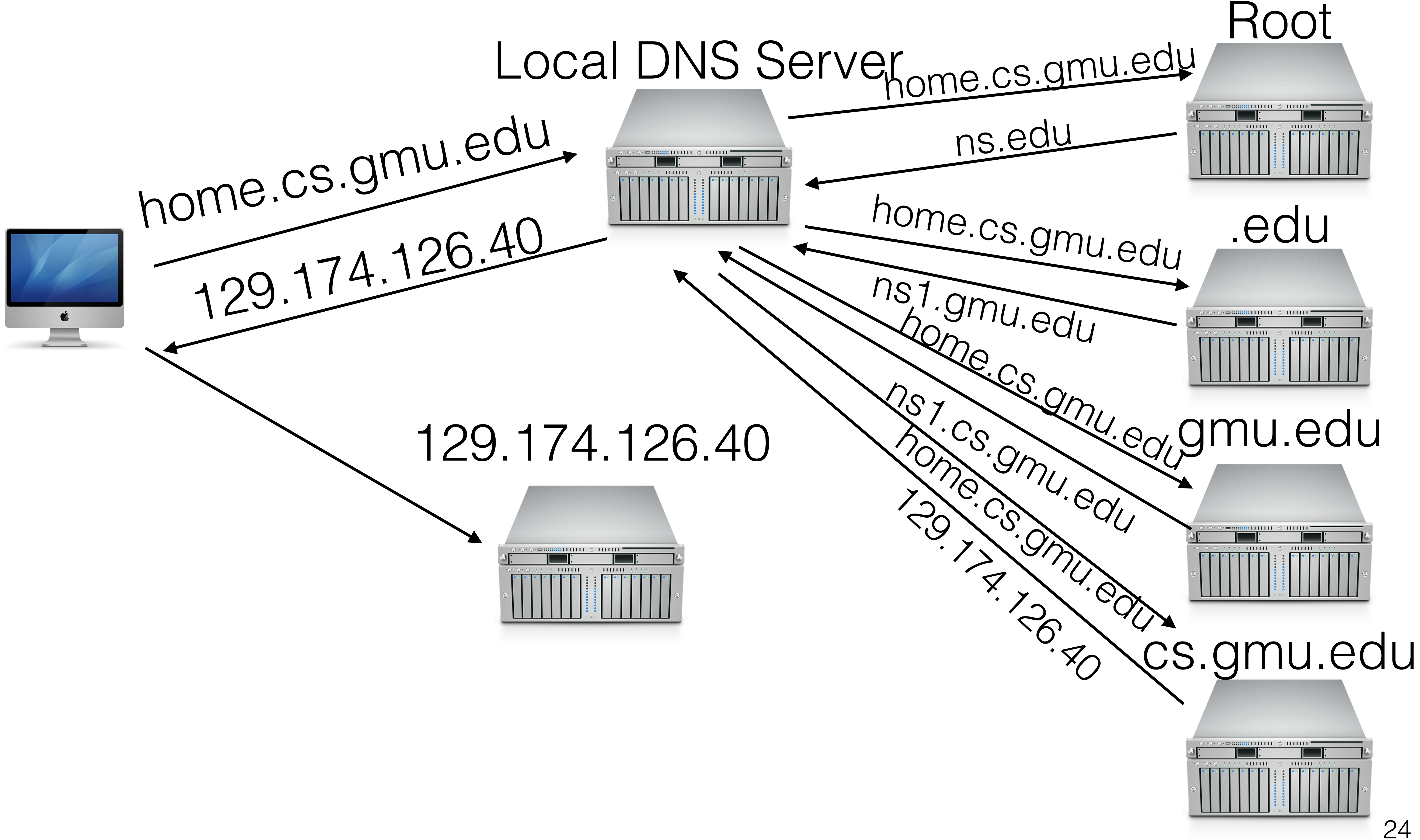


Each zone is a continuous section of name space
Each zone has an associate set of name servers

DNS

- Can have more/less servers replicating each zone based on popularity
- DNS responses are cached at clients
  - Caches periodically time out; bigger zones tend to have longer timeouts
  - Quick response for the same request, also for similar requests

DNS: Example



Local DNS Server

Root

.edu

gmu.edu

cs.gmu.edu

home.cs.gmu.edu

129.174.126.40

home.cs.gmu.edu

ns.edu

home.cs.gmu.edu

ns1.gmu.edu

home.cs.gmu.edu

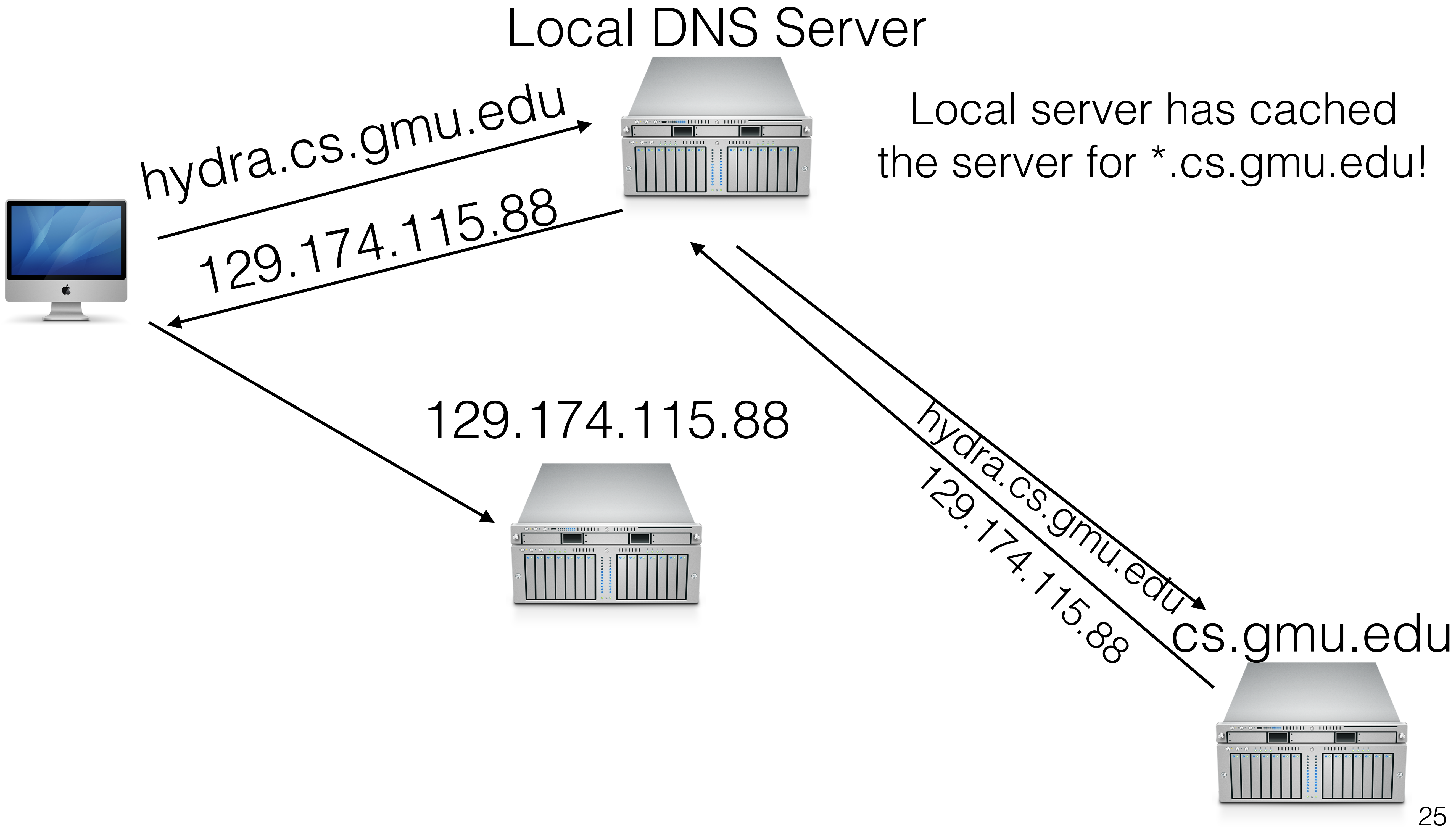
ns1.cs.gmu.edu

home.cs.gmu.edu

129.174.126.40

129.174.126.40

DNS: Example



Local server has cached the server for *.cs.gmu.edu!

DNS

- Trick: can return multiple records for a single query for load balancing
- Large scale distributed database with a hierarchical structure

Hashing

- The hierarchical structure can solve some problems, but not all
- What if we want to make a mechanism to track *pages*, not *domain names*
- Map from a single URL to a set of servers that have that document
- This is the problem attacked by content distribution networks, or CDNs (Akamai, Cloudflare, etc)

CDNs

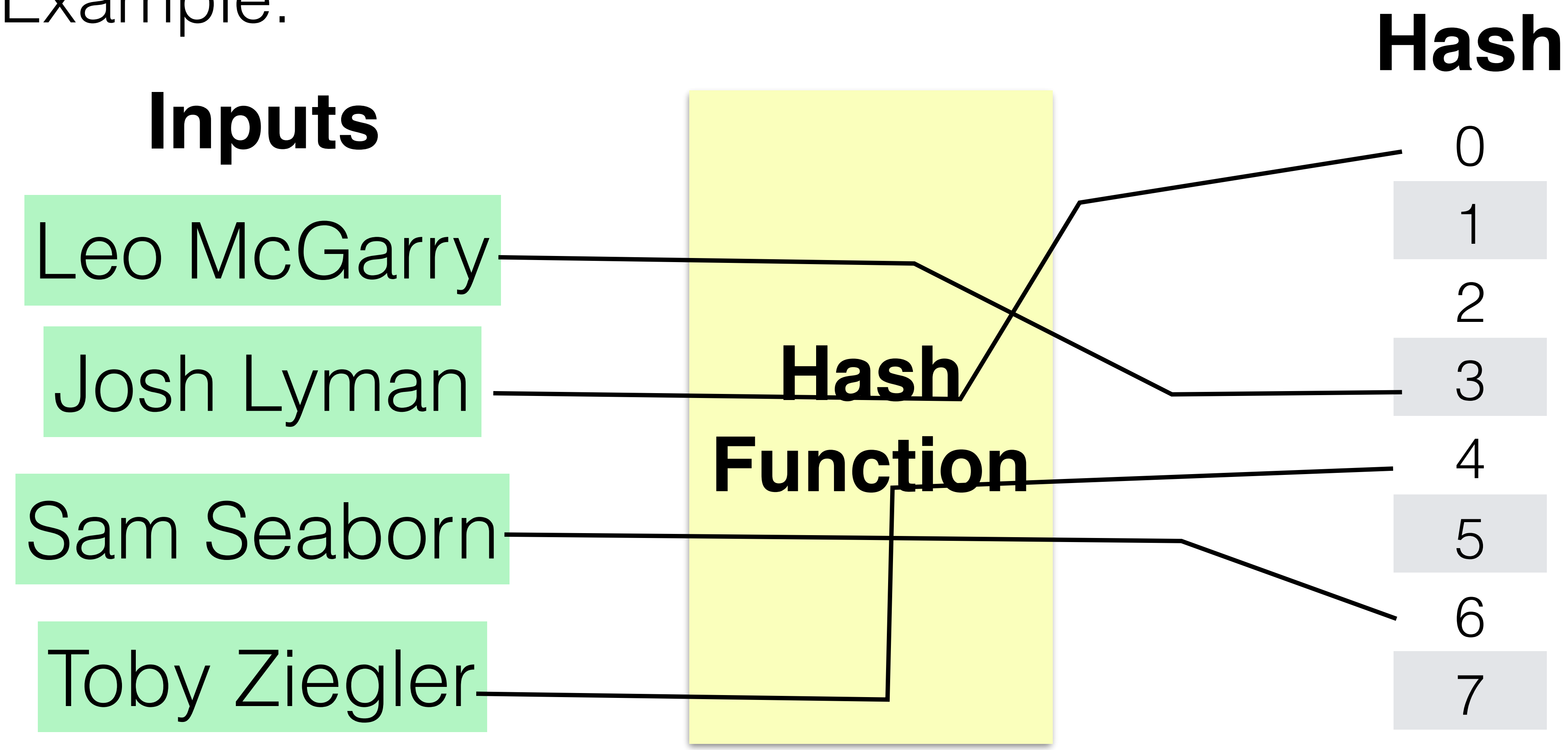
- Idea: create a function hash(key), that for any key returns the server that stores key
- This is called a **hash** function
- Problems?
  - No notion of duplication
  - What if nodes go down/come up?

Hashing

- Input: Some arbitrarily large data (bytes, ints, letters, whatever)
- Output: A fixed size value
- Rule: Same input gives same output, always; "unlikely" to have multiple inputs map to the same output

Hashing

- Compresses data: maps a variable-length input to a fixed-length output
- Relatively easy to compute
- Example:



Inputs

Leo McGarry

Josh Lyman

Sam Seaborn

Toby Ziegler

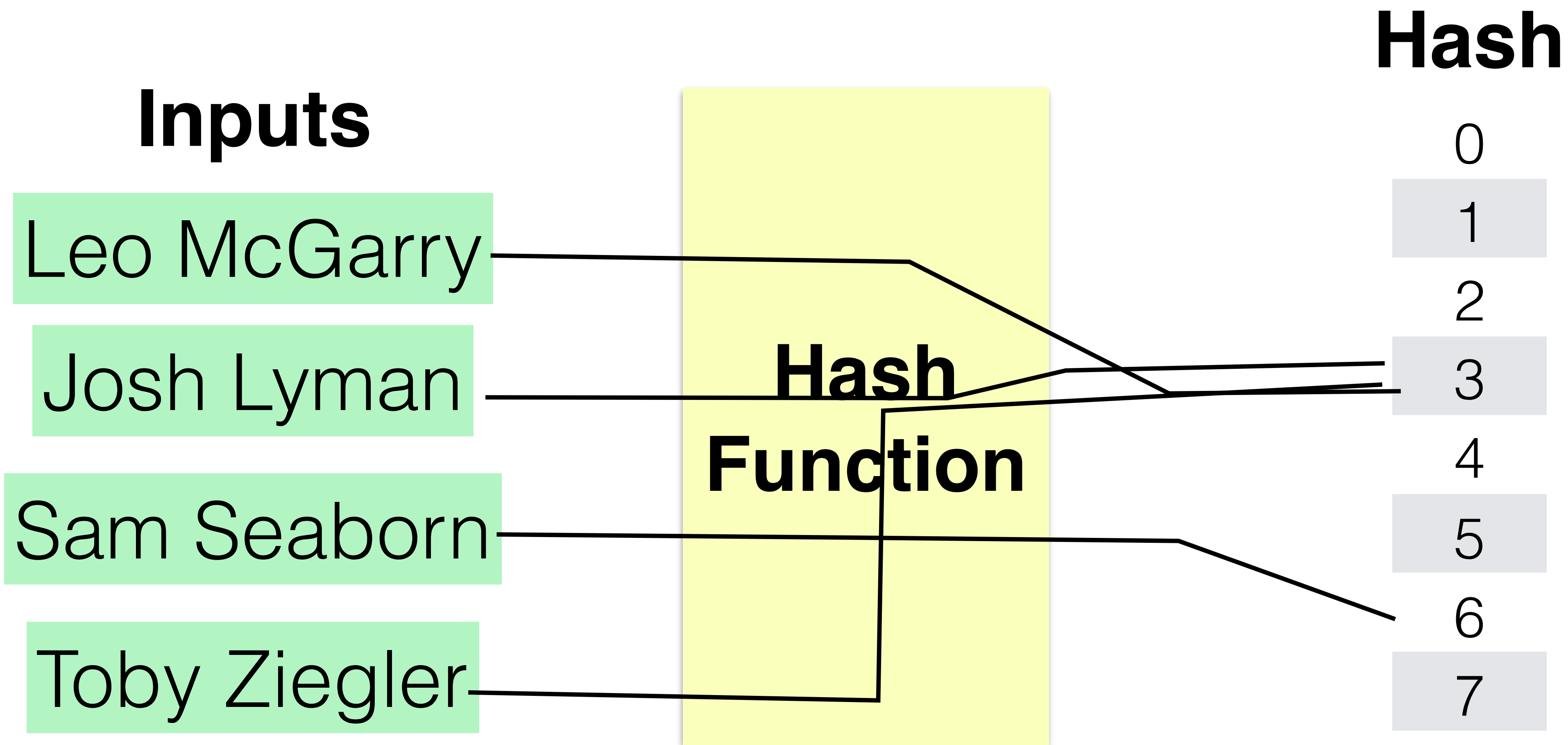
Hash Function

Hash

0
1
2
3
4
5
6
7

Hashing

- The last one mapped every input to a different hash
- Doesn't have to, could be collisions



Hashing

- Hashes have tons of other uses too:
    - Verifying integrity of data
    - Hash table
    - Cryptography
    - Merkle trees (git, blockchain)

Hashing for Partitioning

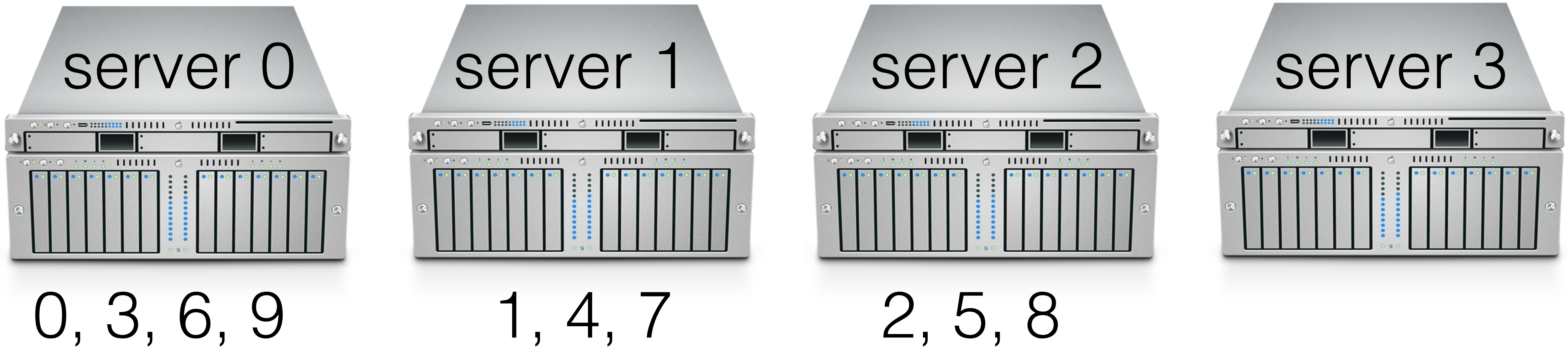
Input		**Hash Result**		**Server ID**
`Some big long piece of text or database key`	*hash()=*	900405	*% 20 =*	5

Conventional Hashing + Sharding

- In practice, might use an off-the-shelf hash function, like sha1
- sha1(url) -> 160 bit hash result % 20 -> server ID (assuming 20 servers)
- But what happens when we add or remove a server?
  - Data is stored on what *was* the right server, but now that the number of servers changed, the right server changed too!

Conventional Hashing

Assume we have 10 keys, all integers



server 0

0, 3, 6, 9

server 1

1, 4, 7

server 2

2, 5, 8

server 3

Adding a new server

Conventional Hashing

Assume we have 10 keys, all integers

0, 4, 8 1, 5, 9 2, 6 3, 7

Adding a new server

8/10 keys had to be reshuffled!
Expensive!

Consistent Hashing

- Problem with regular hashing: very sensitive to changes in the number of servers holding the data!
- Consistent hashing will require on average that only K/n keys need to be remapped for K keys with *n* different slots (in our case, that would have been 10/4 = 2.5 [compare to 8])

Consistent Hashing

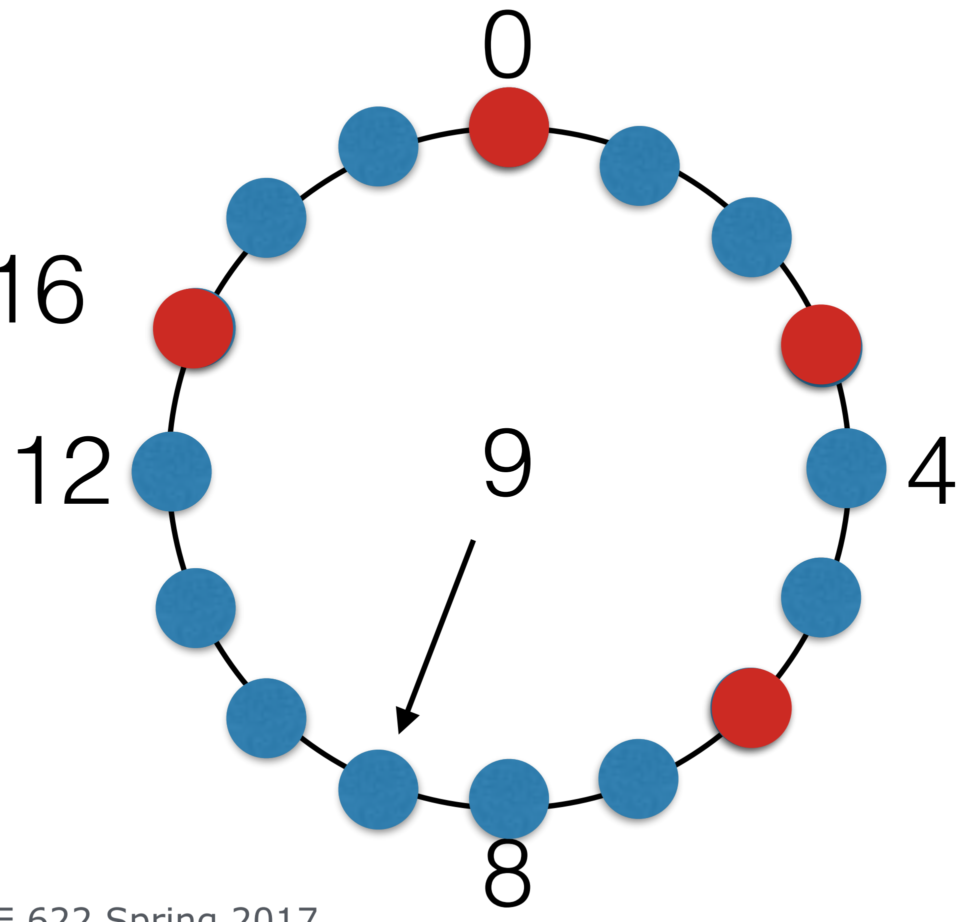
- Construction:
    - Assign each of C hash buckets to random points on mod 2^n circle, where hash key size = n
    - Map object to pseudo-random position on circle
    - Hash of object is the closest clockwise bucket

Example: hash key size is 16

Each 🔵 is a value of hash % 16

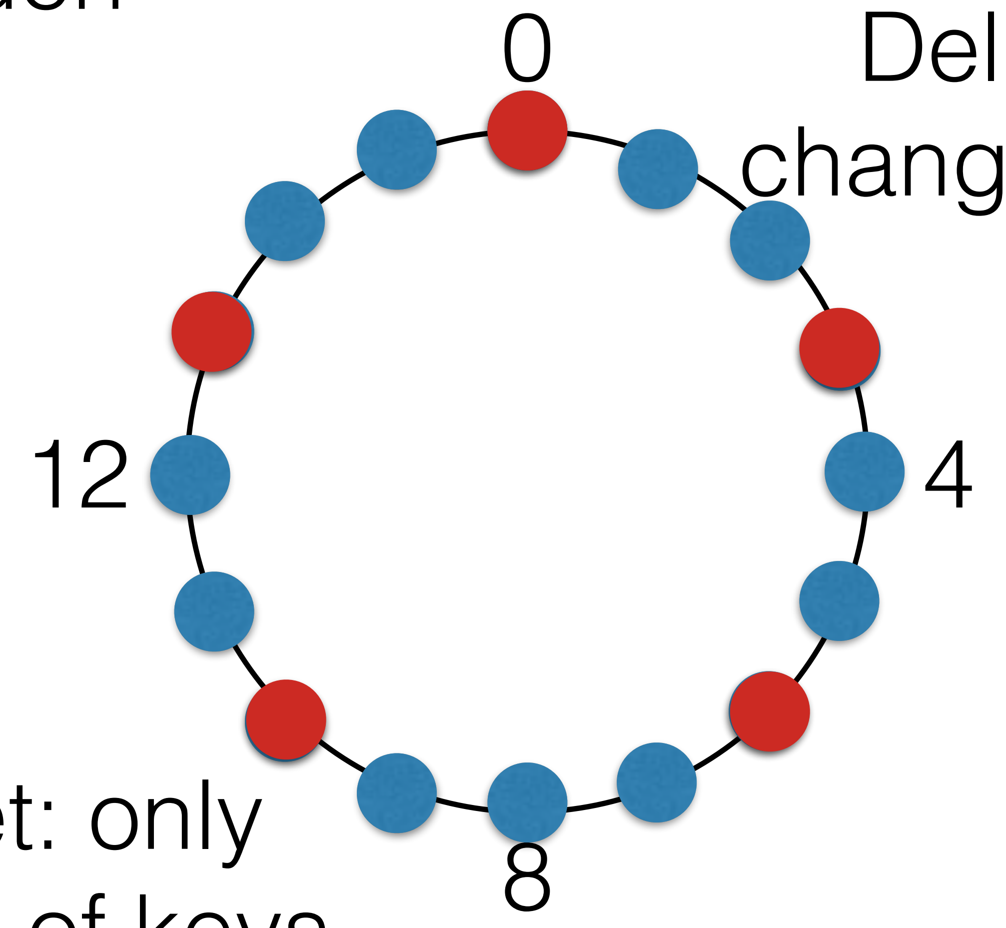
Each 🔴 is a bucket

Example: bucket with key 9?



Consistent Hashing

It is relatively smooth: adding a new bucket doesn't change that much

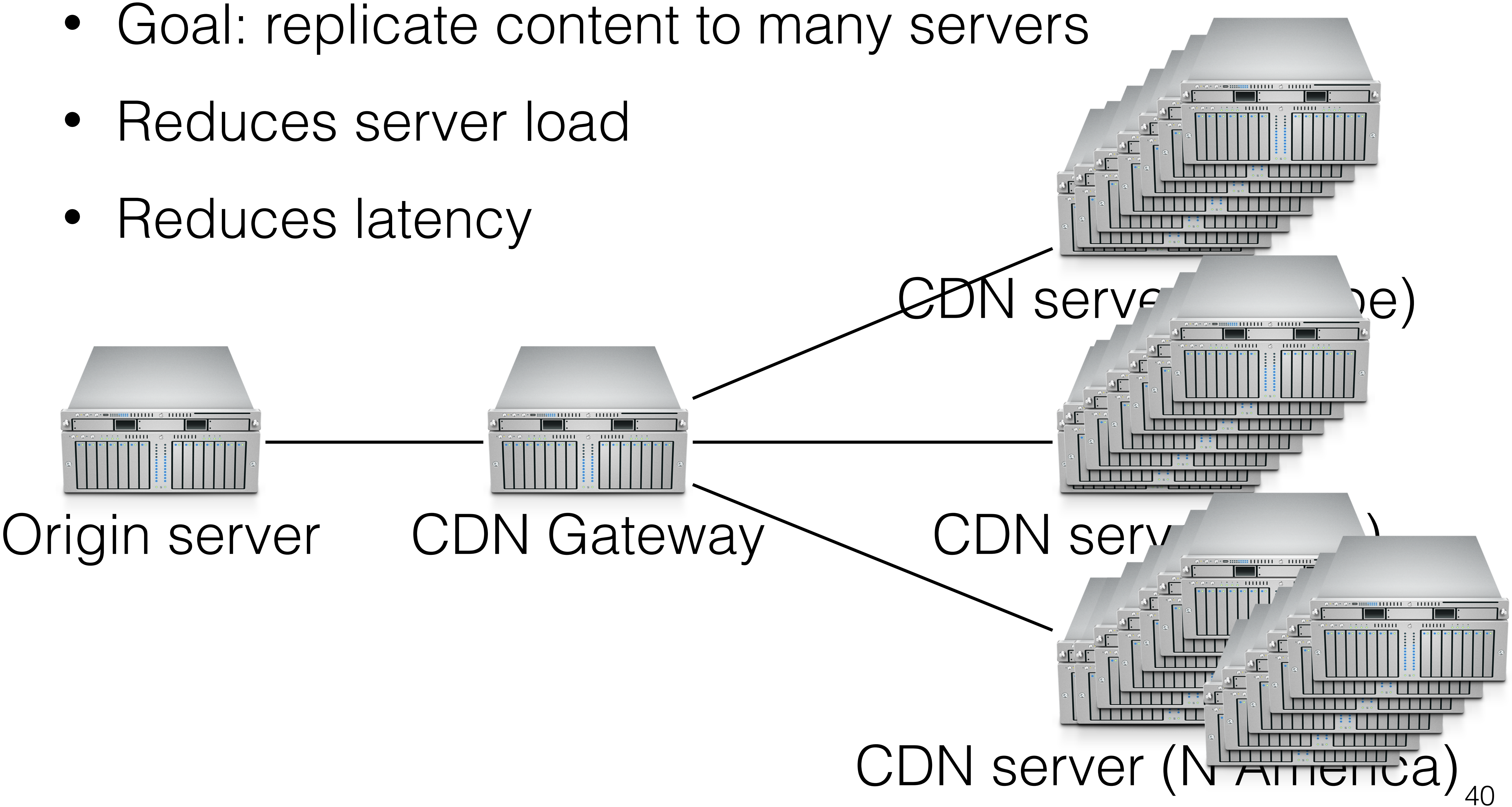


Delete bucket: only changes location of keys 1,2,3

Add new bucket: only changes location of keys 7,8,9,10

Consistent Hashing in Practice (CDN)

- Goal: replicate content to many servers
- Reduces server load
- Reduces latency



CDN Challenges

- How do we replicate the content?
  - Assume: only static content
- Where do we replicate the content?
  - Assume: infinite money
- How to choose amongst known replicas?
  - Lowest load? Best performance?
- How to find the replicated content?
  - Tricky

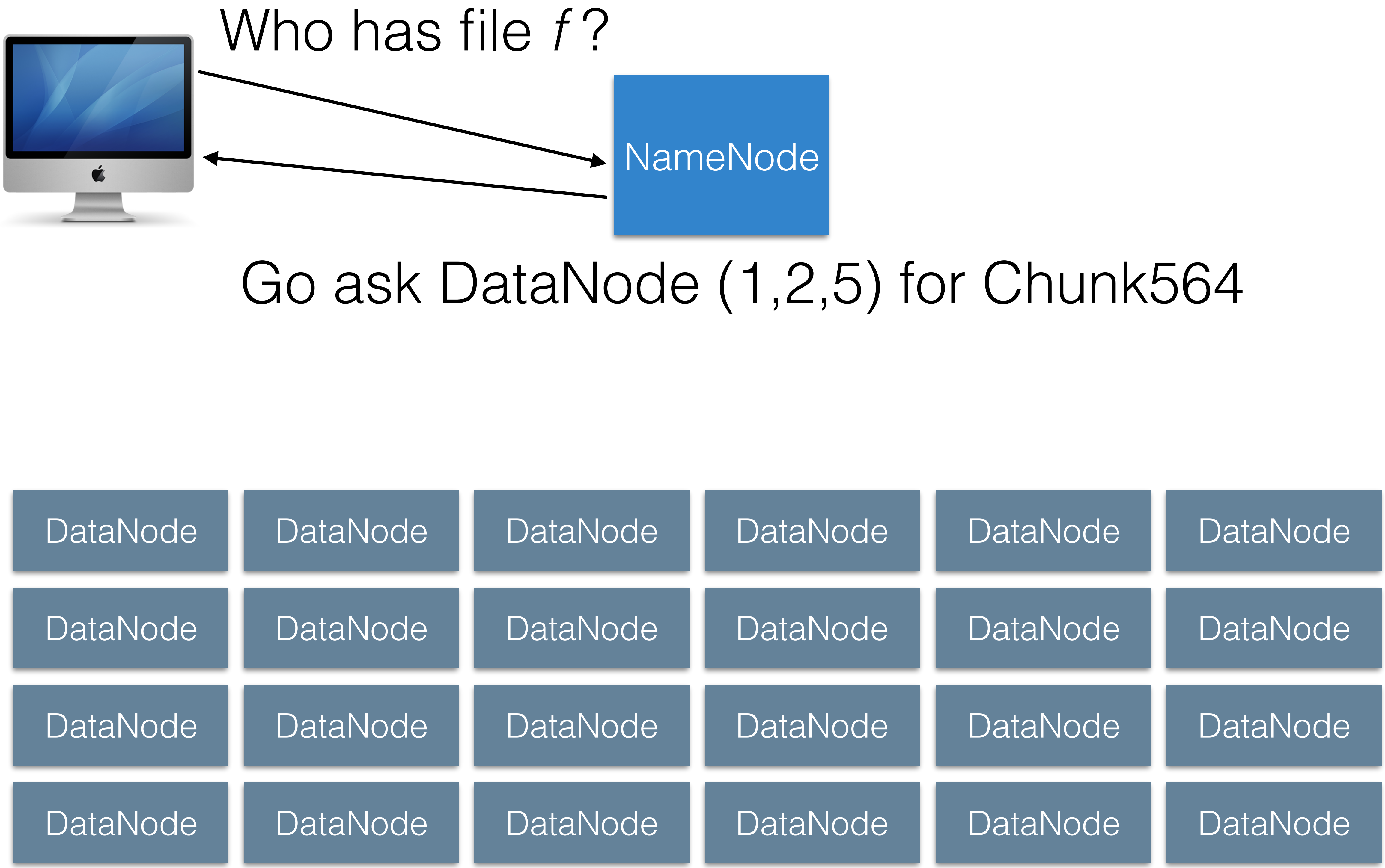
Finding the replicas

- Maintain 1000’s of data centers across the internet, each with many servers
- Hash(URL’s domain) maps to a server
- Akamai maintains their own DNS infrastructure, with two tiers (global and regional)
- Lookup apple.com -> akamai.net -> g.akamaitech.net -> actual cache server

Finding the replicas

- Lookup apple.com -> akamai.net -> g.akamaitech.net -> actual cache server
- The address returned for g.akamaitech.net is one that is near the client (geographically)
- The address returned by that regional server is determined by which local server has the content
  - As determined by consistent hashing

P2P

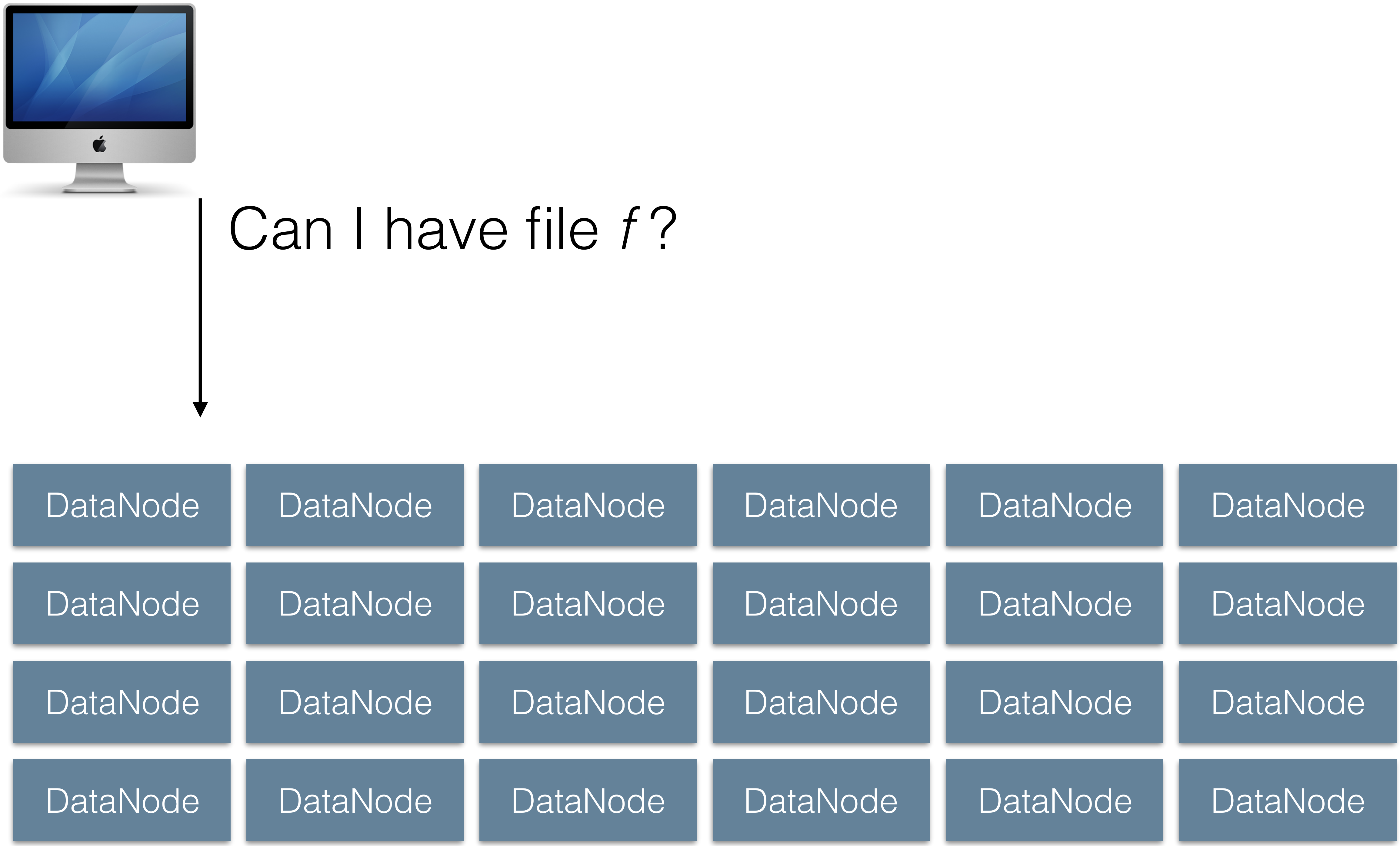


Who has file *f* ?

NameNode

Go ask DataNode (1,2,5) for Chunk564

P2P



Can I have file *f* ?

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

DataNode DataNode DataNode DataNode DataNode DataNode

P2P

- Goal: IF there must be a master, all that it knows is the address of a few clients using the system
- Otherwise, everyone talks to each other, figures it out
- Replicate files, store them on clients, let clients find files from each other
- Challenges:
  - Where to find data?
  - What to do when clients come and go?

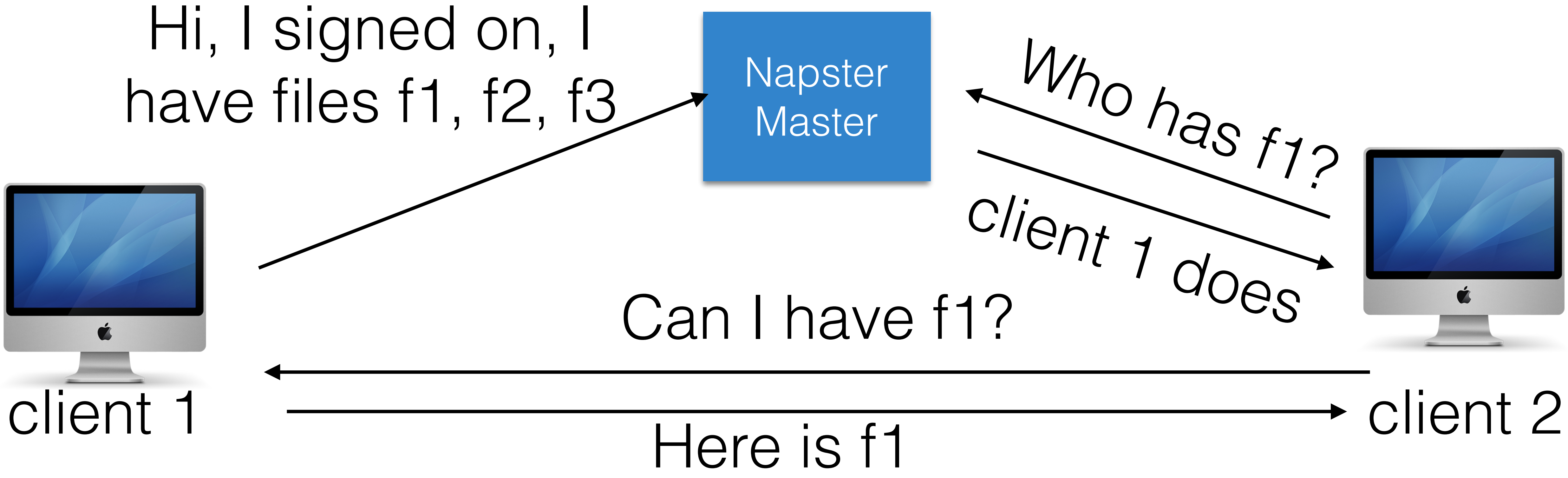
P2P

- Break it down into four operations:
  - **Join** the network and begin participating
  - **Publish** a file to the network, letting others know you have it
  - **Search** for a file that you want
  - **Fetch** a file once it is found

Napster

- Single master (centralized DB) stores metadata and client status
- **Join**: Client contacts master
- **Publish:** Client reports list of files to master
- **Search:** Query the server, find who has the file you want
- **Fetch:** Get directly from that peer client

Napster



Hi, I signed on, I have files f1, f2, f3

Napster Master

Who has f1?

client 1 does

Can I have f1?

Here is f1

client 1

client 2

Doesn't everything just look like GFS, even things that predated it? :)

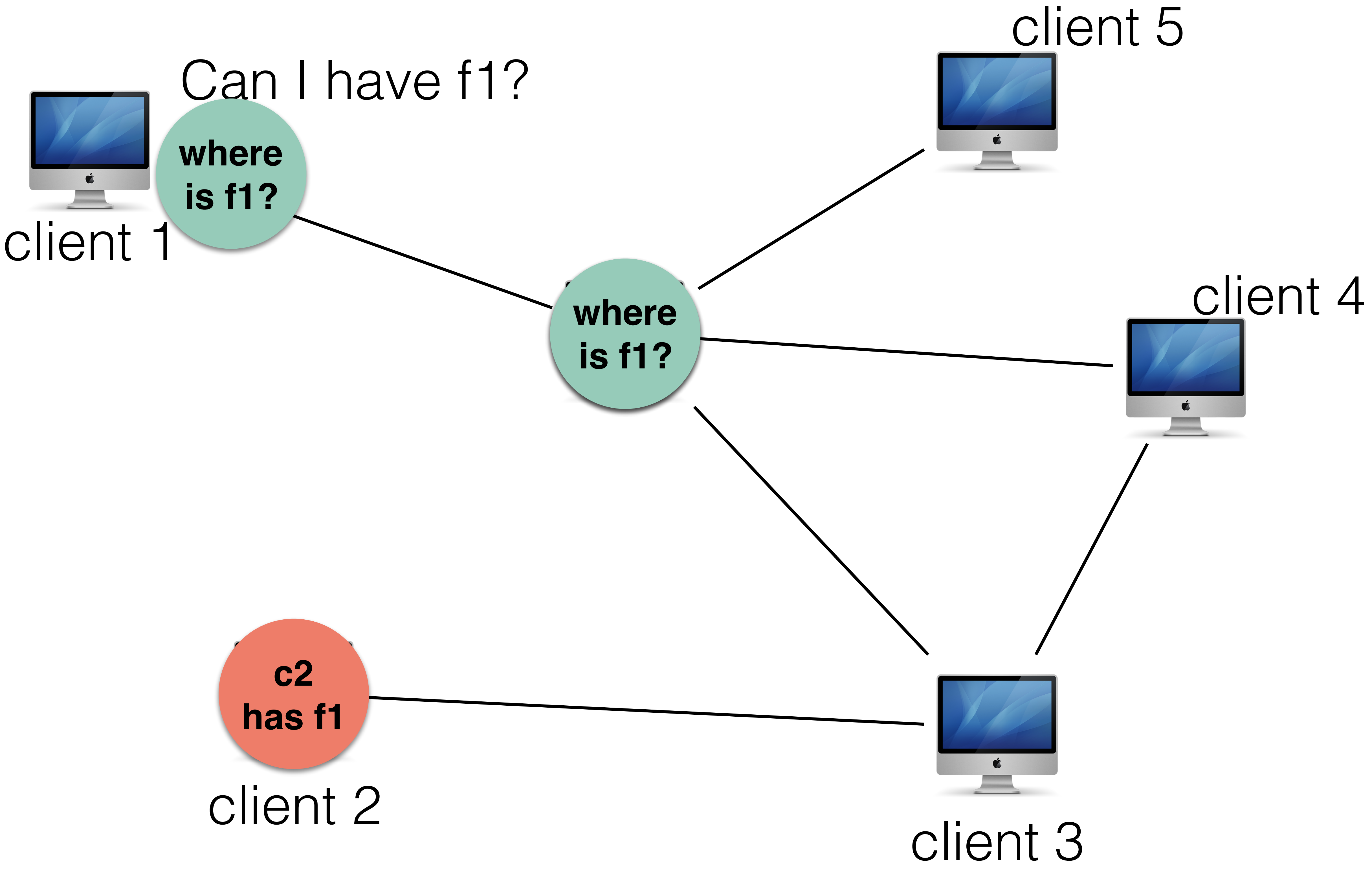
Napster

- The good:
  - Simple
  - Finding a file is really fast, regardless of how many clients there are - master has it all
- The bad:
  - Server becomes a single point of failure
  - Server does a lot of processing
  - Server having all of metadata implies significant legal liabilities

Gnutella 1.0

- **Join:** Client contacts a few other clients to find “neighbors”
  - Requires some initial mechanism to bootstrap
- **Publish:** N/A
- **Search:** Client asks neighbors for file, who ask their neighbors for file, who asks their neighbors out to some depth
- **Fetch:** Clients directly communicate with each other

Gnutella 1.0


client 5
Can I have f1?
where is f1?
client 1
where is f1?
client 4
c2 has f1
client 2
client 3

Gnutella

- This is called "flooding"
- Cool:
  - Fully decentralized
  - Cost of search is distributed - no single node has to search through all of the data
- Bad:
  - Search requires contacting many nodes!
  - Who can know when your search is done?
  - What if nodes leave while you are searching?

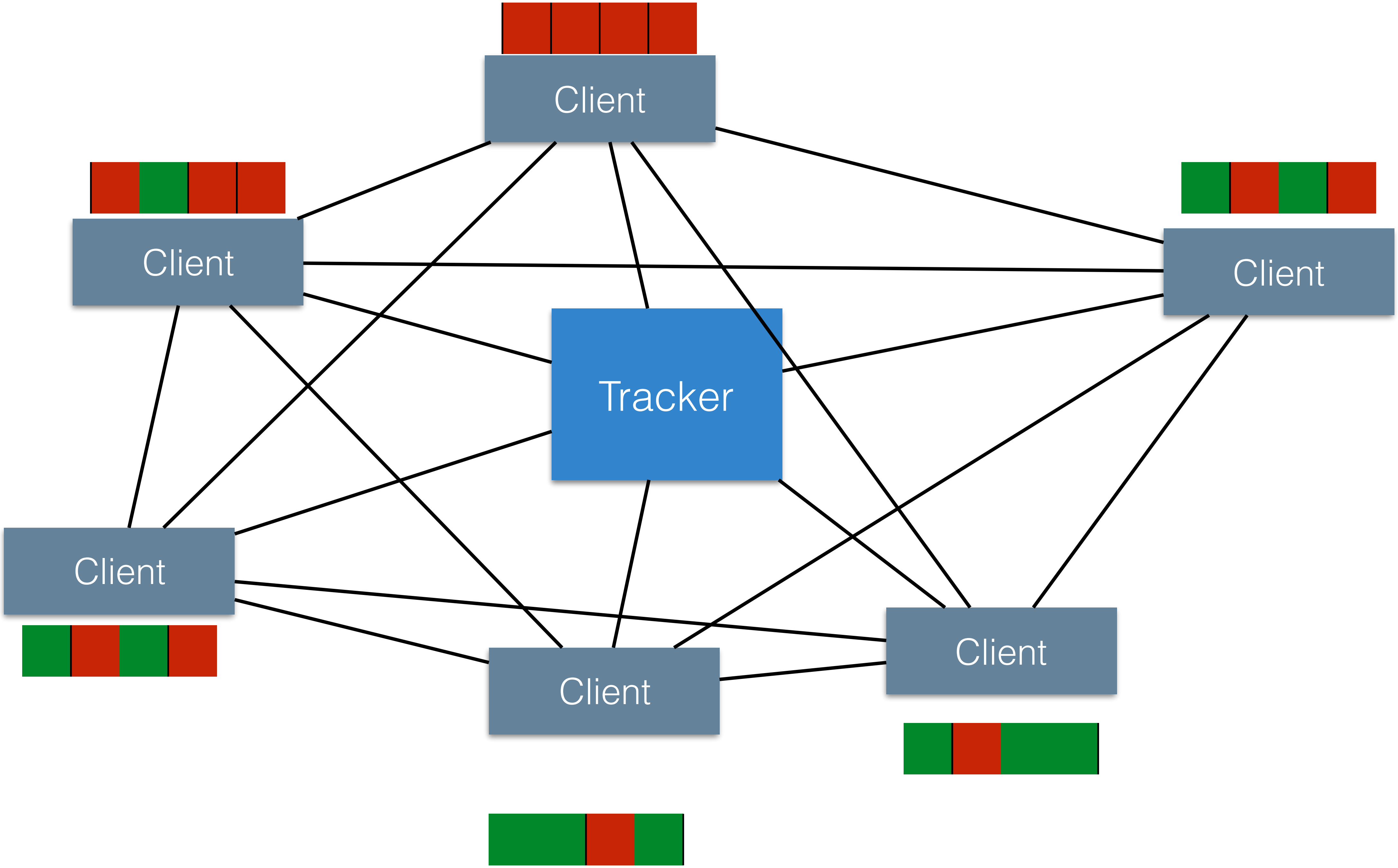
BitTorrent

- "Swarming"
- **Join**: Contact master "tracker," get list of peers
- **Publish**: Run a tracker server
- **Search**: Out-of-band (e.g. google)
- **Fetch**: Download chunks of files from peers

BitTorrent vs Napster

- Focus on **less** files, each of which is **larger**
- Files are broken into chunks -> can get different pieces of a file from different clients
- Anti-freeloading mechanisms - if you don't share, you don't get to play!
  - Since a big file is many chunks, once you get a chunk you can immediately share it with others
- Trackers are still single-points of failure, but assumption is 1 tracker per file

BitTorrent



BitTorrent

- "Tit-for-tat" sharing strategy
- A is getting data from B, C, D
  - A will let the fastest of those get data from A
  - A will be optimistic though, and let nodes who haven't shared anything yet have some data so that they can have a chance to share

DHT (Distributed Hash Table)

- Goal:
  - Guarantee that a file is always found within some bounded and reasonable number of steps
- Abstraction:
  - Create a lookup table, mapping from file to node that has that file (much like Napster)
  - BUT distribute this lookup table amongst the nodes participating (no single master)

DHT

- **Join:** Contact some other node to bootstrap: integrate yourself into the DHT, get a node ID and list of participating nodes
- **Publish:** Tell "mostly the correct" node that you have a file
- **Search:** Query for a file, asking first a "mostly correct" node
- **Fetch:** Contact node that has it directly
- How do we know where to route? Consistent hashing!

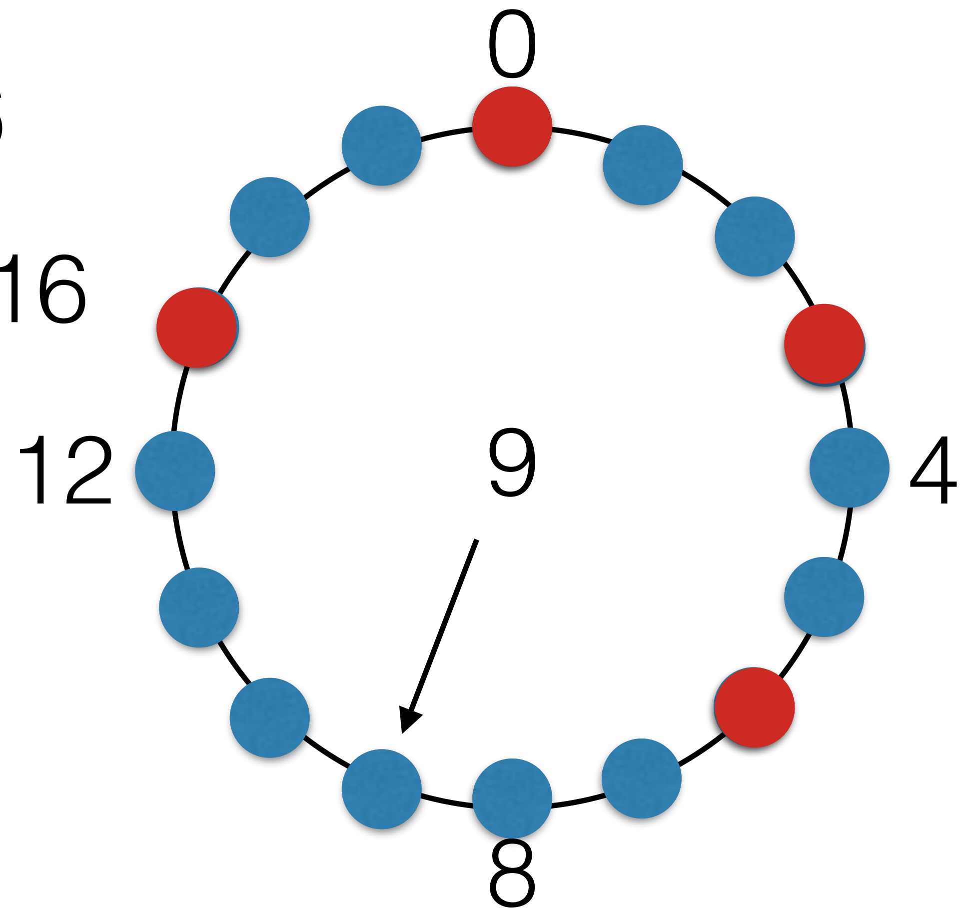
Reminder: Consistent Hashing

Example: hash key size is 16

Each 🔵 is a value of hash % 16

Each 🔴 is a bucket

Example: bucket with key 9?



DHT

- Pros:
  - Guarantees that if the data is in the network, you'll find it in log(n) time (compare to Gnutella - pseudo-random search)
  - Good for caching, infrequently written data
- Cons:
  - Can really only match on exact keys
  - The node join/leave story is really bad - if we are distributed across the internet, a node leaving/joining might involve moving hundreds of GBs around

DHT Applications

- Use a DHT instead of a tracker for BitTorrent!
- Bootstrap: find a DHT peer
- Application: As you acquire files or look for files, add those facts into the DHT

Lab: Consistent Hashing

- Let’s stop it with the replication for a bit and focus on partitioning (sharding)
- Now we’ll have 5 Redis servers, and partition our data set between them.
- Use consistent hashing to pick which server to use to access data