# Sharding & CDNs

CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

# Review: Distributed File Systems

- Challenges:
  - Heterogeneity (different kinds of computers with different kinds of network links)
  - Scale (maybe lots of users)
  - Security (access control)
  - Failures
  - Concurrency

# Review: NFS

- Cache file blocks, file headers, etc., at both clients and servers.
- Advantage:  No network traffic if open/read/write/close can be done locally.
- But: failures and cache consistency.
- NFS trades some consistency for increased performance... let’s look at the protocol.

# Review: NFS + Server crash?

- Data in memory but not disk lost
- So... what if client does seek() ;  /* SERVER CRASH */; read()
  - If server maintains file position, this will fail.  Ditto for open(), read()
- Stateless protocol:  requests specify exact state. read() -> read( [position]).  no seek on server.

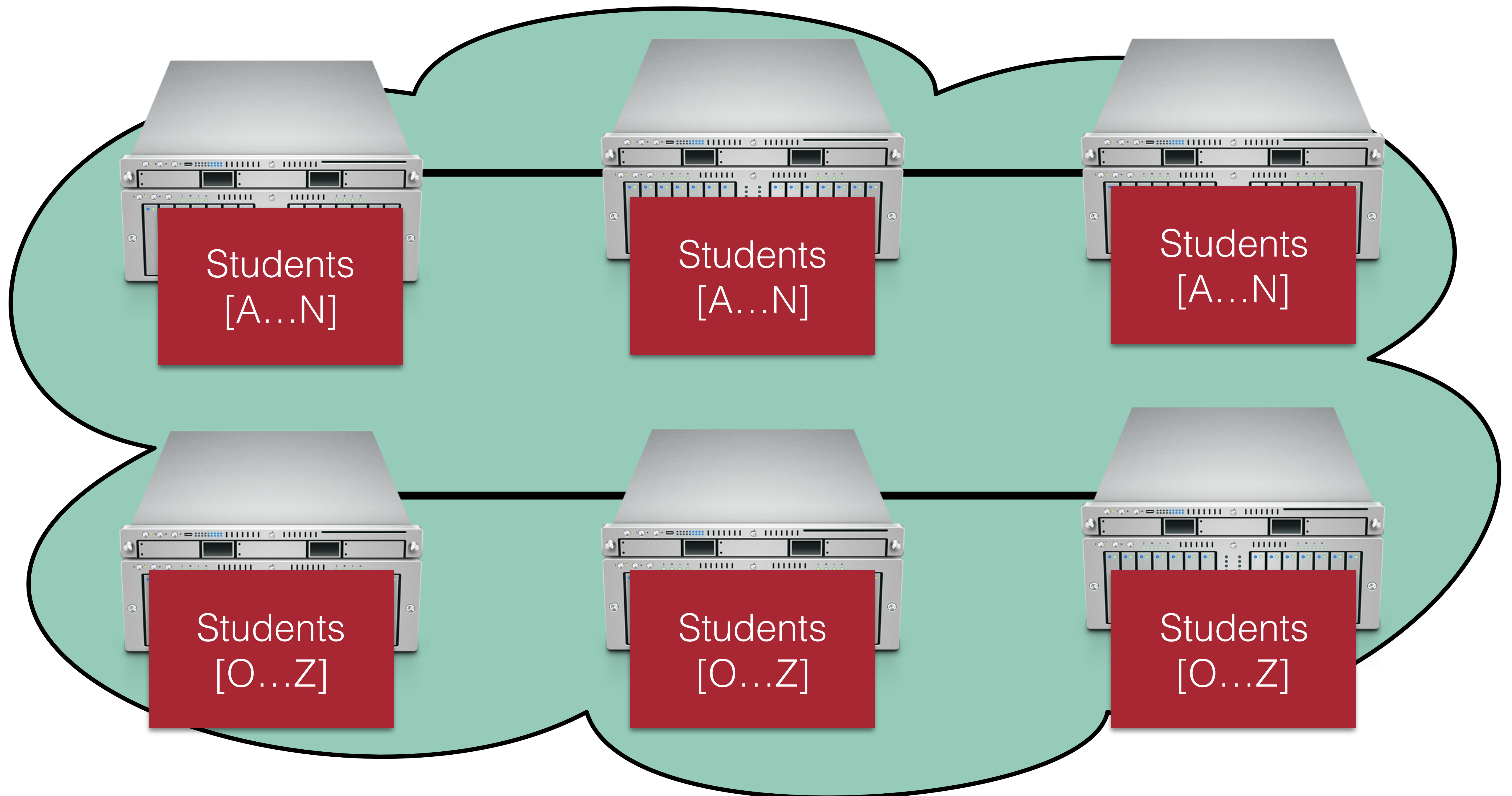
# Review: NFS Caching - Close-to-open

- Implemented by most NFS clients
- Contract:
  - if client A write()s a file, then close()s it,
  - then client B open()s the file, and read()s it,
  - client B's reads will reflect client A's writes
- Benefit: clients need only contact server during open() and close()—not on every read() and write()

# Announcements

- HW3 is out!
  - http://www.jonbell.net/gmu-cs-475-spring-2018/homework-3/
- Midterm: Wednesday
- Today: Sharding and CDNs
  - How do we find out where to find data?
- Additional reading: OS TEP Ch 49

# Partitioning (Sharding) + Replication



# Partitioning (Sharding) + Replication

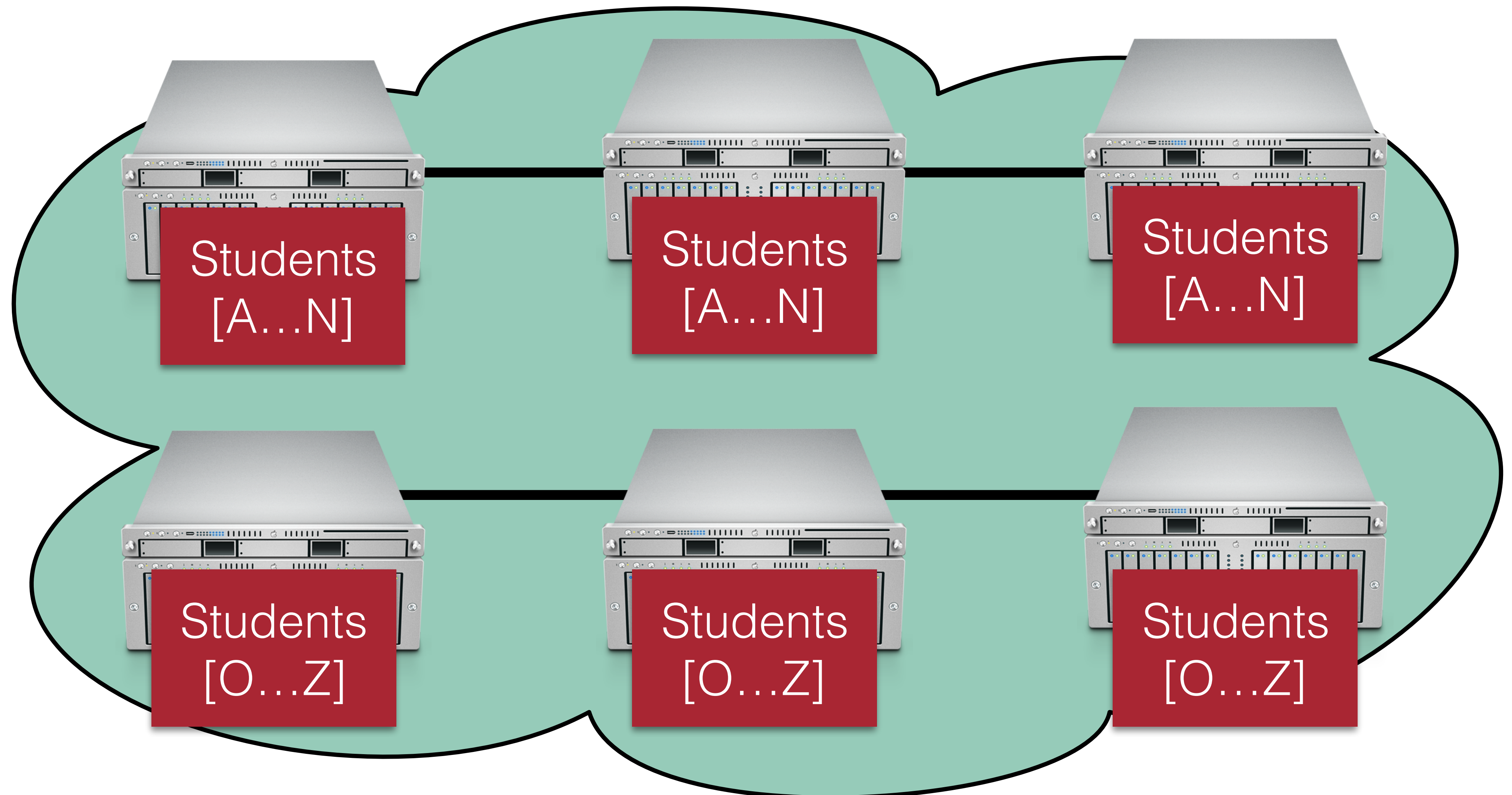
- We can solve our **discovery** problem if we can define a **consistent** way to store our data and share those rules
- Create "buckets," and use a "hash function" to map from a key to a bucket
- Example: All files starting with the letter "A" are stored on servers 1,2,3; all files starting with the letter "B" are stored on severs 4,5,6, etc.

# Strawman Sharding Scheme

**In this class:**

**40 students**

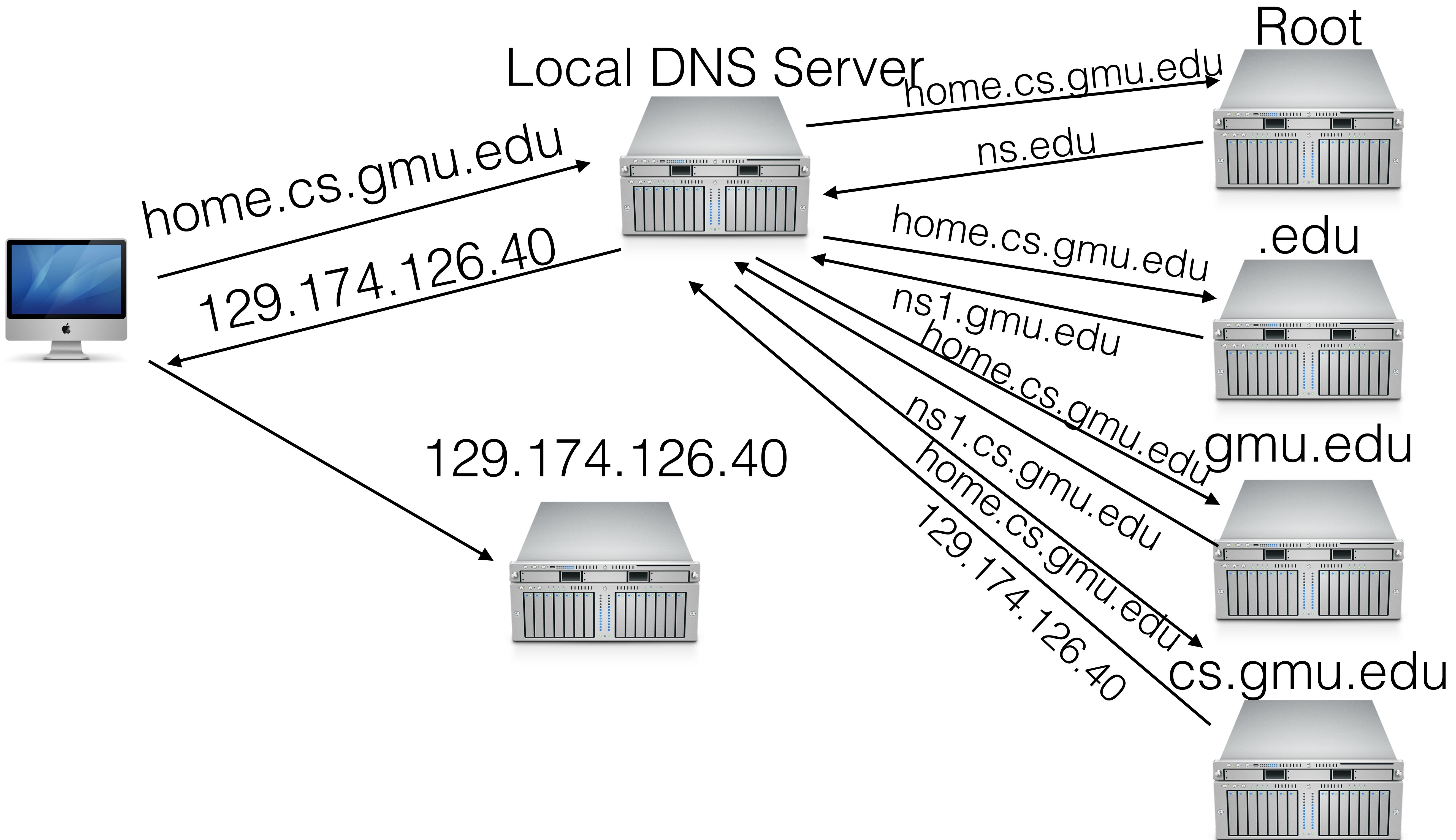
**16 students**



# 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/cs475 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: Example



# DNS

# 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)

# Hashing

- BitTorrent & many other modern p2p systems use content-based naming
- Content distribution networks such as Akamai use consistent hashing to place content on servers
- Amazon, Linkedin, etc., all have built very large-scale key-value storage systems (databases--) using consistent hashing

# *Hashing*

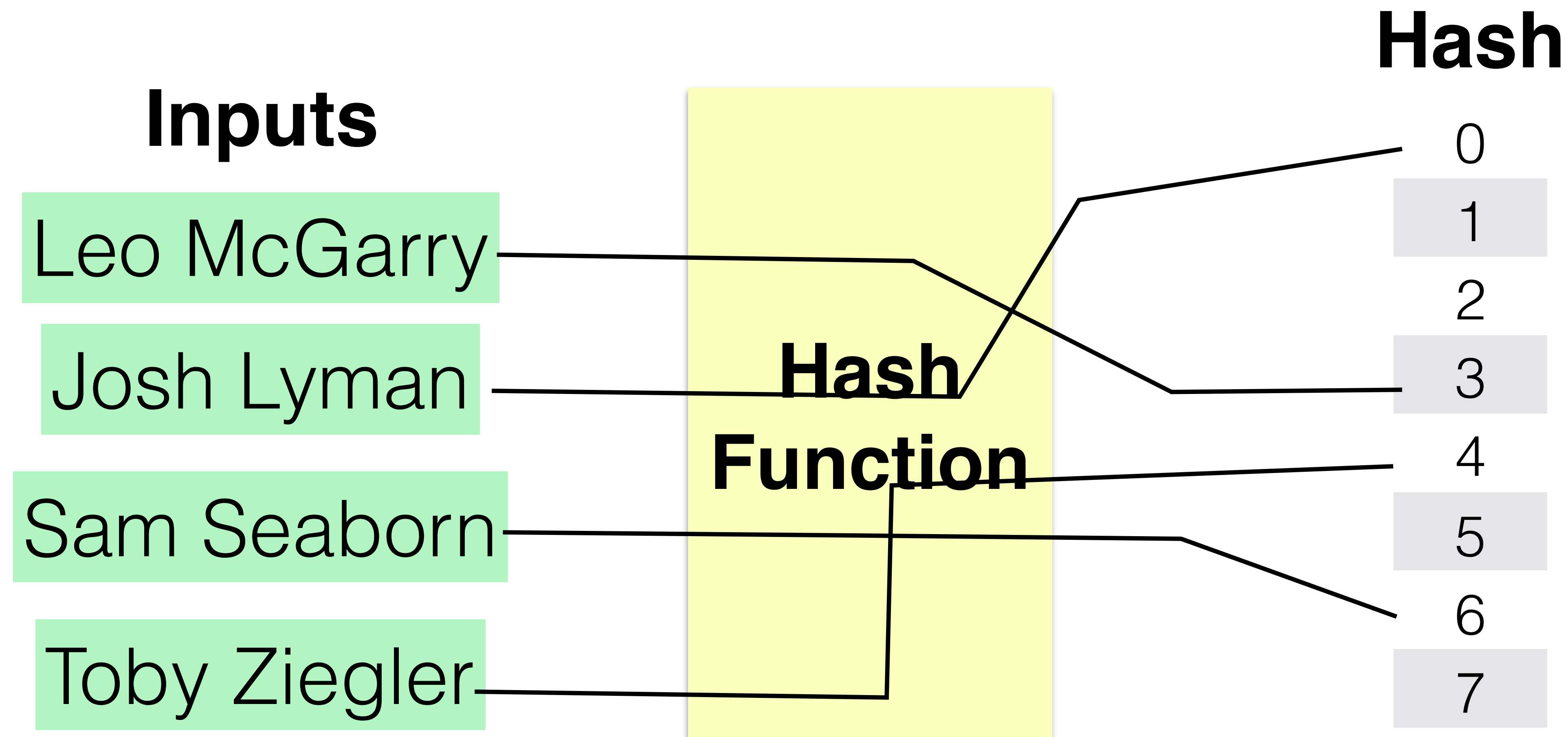
- *Idea: create a function hash(key), that for any key returns the server that stores key*
- This is called a **hash** function
- Problems?
  - No notion of duplication (what if a server goes down?)
  - What if nodes go down/come up?

# 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

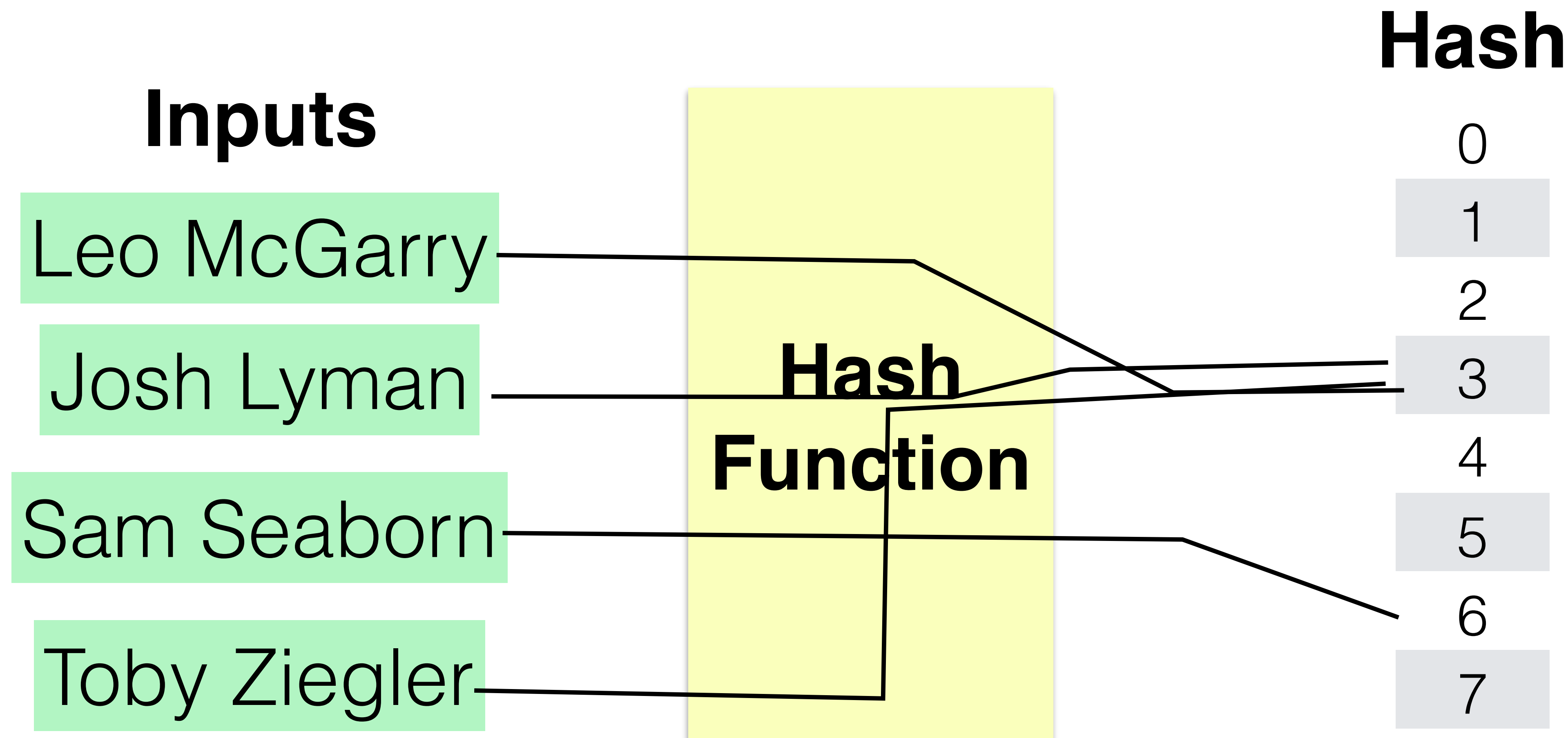
# Hashing

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



# 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

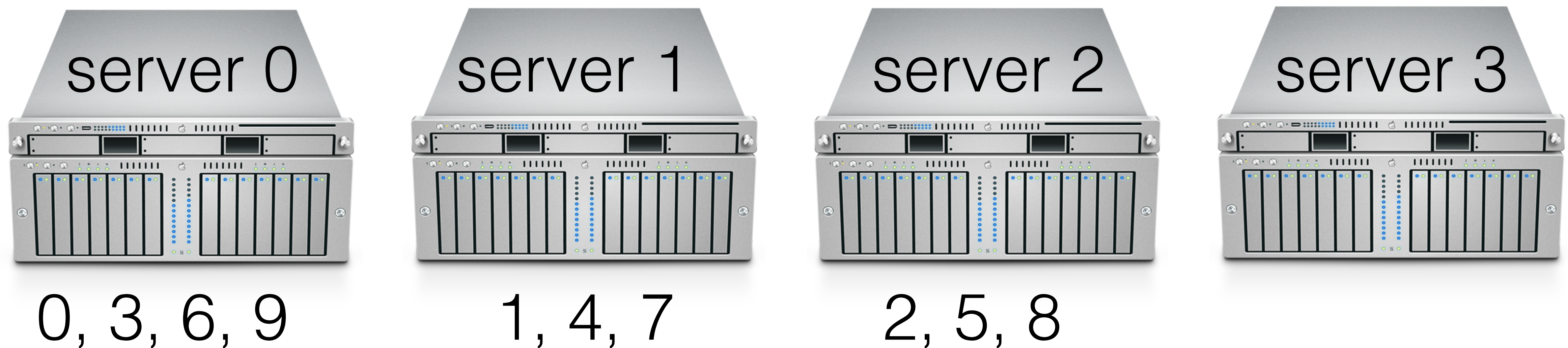
| **Input** | | **Hash Result** | | **Server ID** |
|---|---|---|---|---|
| `Some big long piece of text or database key` | *hash()=* | `900405` | *% 20 =* | `5` |

# 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

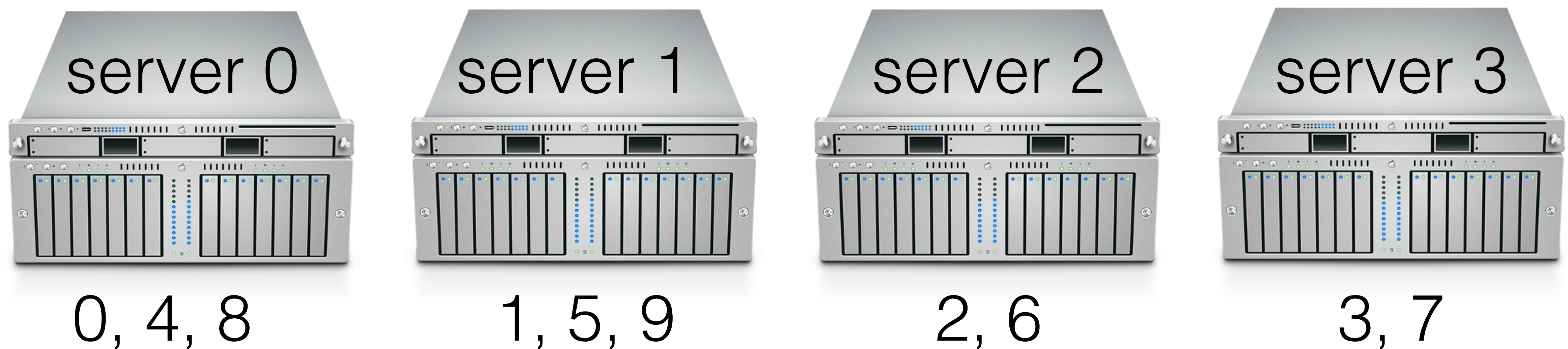
Assume we have 10 keys, all integers




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

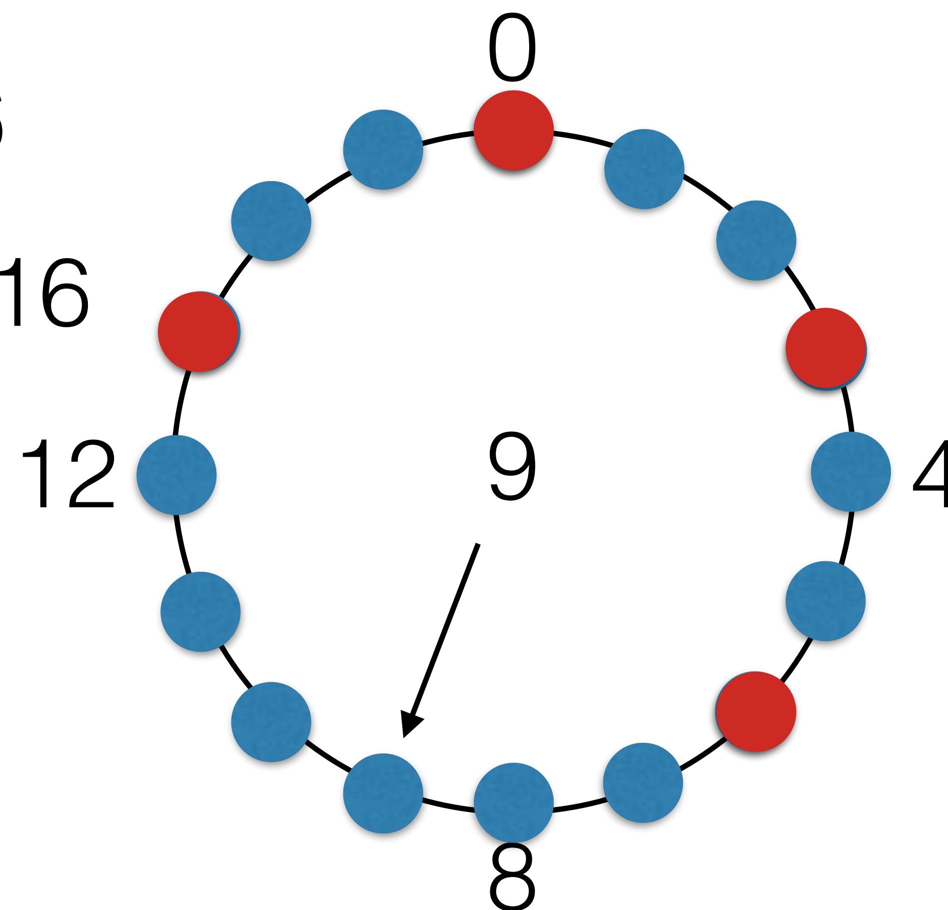
- Construction:
  - Assign each of C hash buckets to random points on mod $2^n$ circle, where hash key size = n
  - Map object to pseudo-random position on circle
  - Hash of object is the closest clockwise bucket

Example: hash key size is 16

Each ● is a value of hash % 16

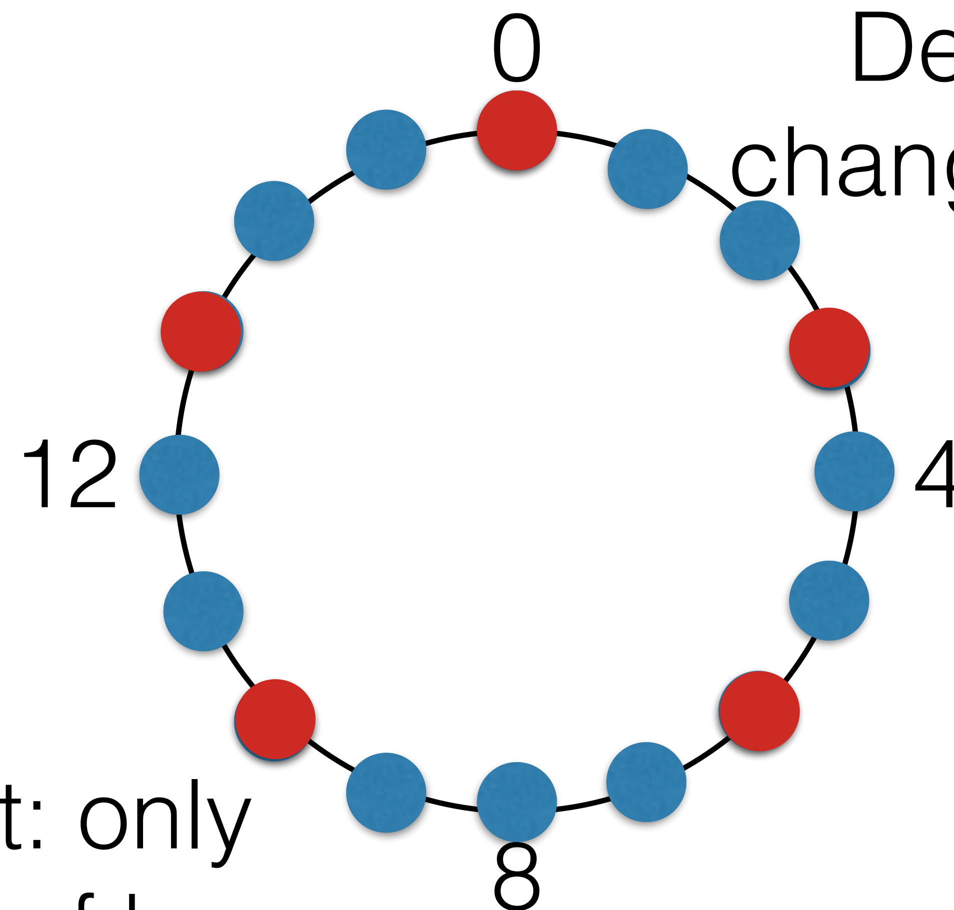
Each ● is a bucket

Example: bucket with key 9?



# Consistent Hashing

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

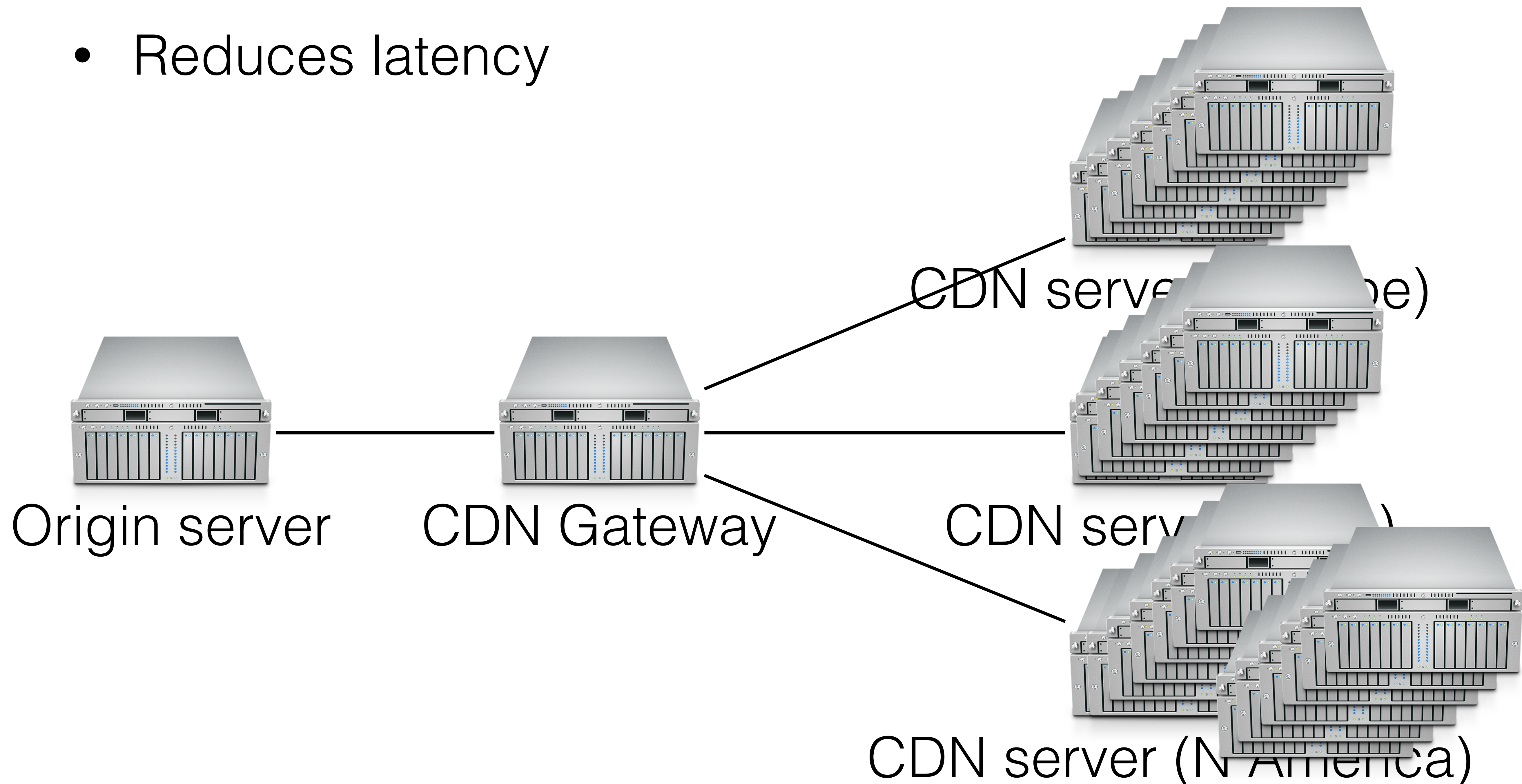



Delete bucket: only changes location of keys 1,2,3

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

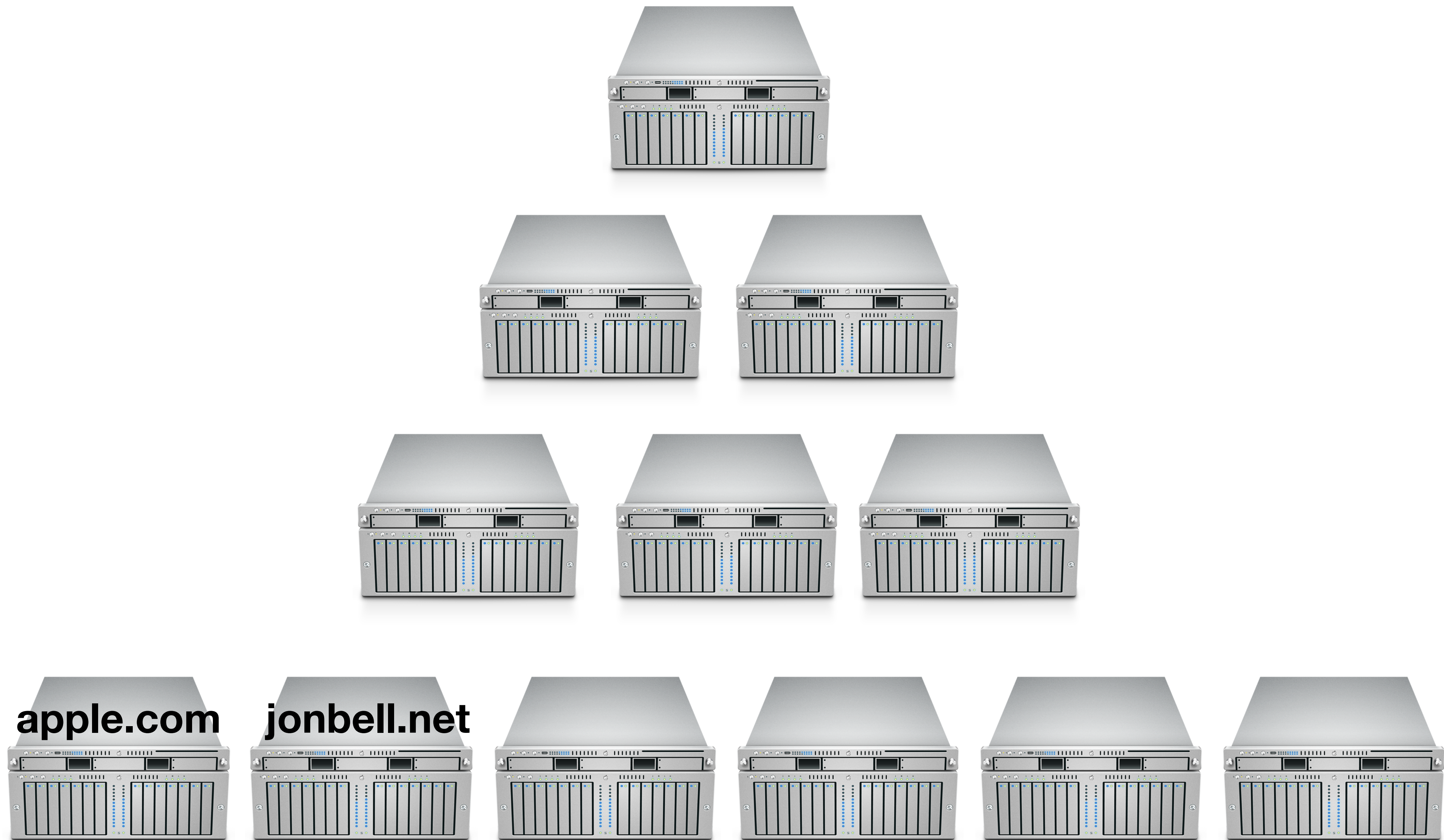
# CDN Challenges Finding Content

- Desire super, super low latency
- Serving a single request is probably very very cheap (e.g. read a 1KB file from memory/SSD and return it)
- But we want to serve billions of these requests at once
- VERY important that we can route requests fast

# CDN: How to find content?

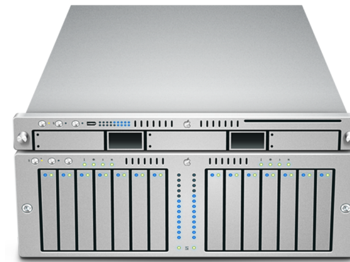
**Problem: how the heck do we figure out what data should go where?**

# CDN: How to find content?




**Problem: How do we organize the tiers/shortcut from URLs to servers?**
**1 server per domain doesn't help balance load**

# CDN: How to find content?



**Master server directs all requests to appropriate cache server**

**Big cluster of cache servers**

**Problem: Master becomes a huge bottleneck**
**Millions of requests, each request needs to be processed incredibly fast**

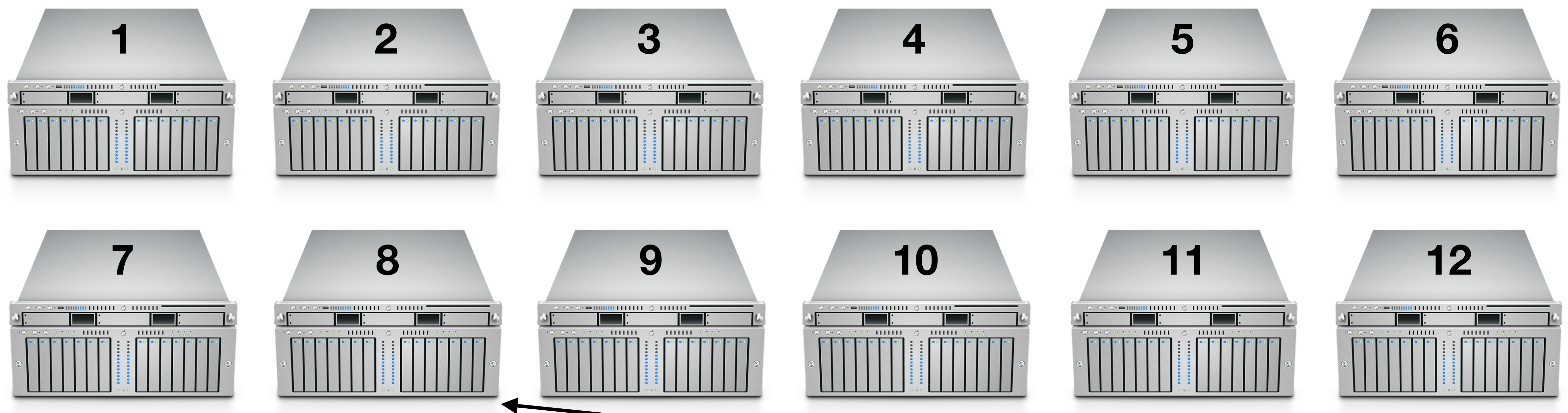
# Finding the replicas

- 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

# CDN: Finding Content




**Consistent hash(**http://www.jonbell.net/gmu-cs-475-spring-2018/homework-3/**)=8**