# Distributed Systems 2: Distributed Key-Value Stores

**Sam Kumar**

**CS 162: Operating Systems and System Programming**

**Lecture 24**

**https://inst.eecs.berkeley.edu/~cs162/su20**

Read: Distributed Systems for Fun and Profit, Ch 4

# So, what do you need to know about networking???

- Layered Internet architecture (distributed system design)
  - The five layers and the role of each layer
  - Significance of IP as the “narrow waist”
  - How it embodies the end-to-end principle

- TCP (role of OS)
  - Role of sequence numbers and ACKs
  - Buffering (the four buffers, and when data is added to/removed from each)
  - Flow control: advertised window size and how it is used
  - Congestion control: what it is, not how it works

- Any questions about these?

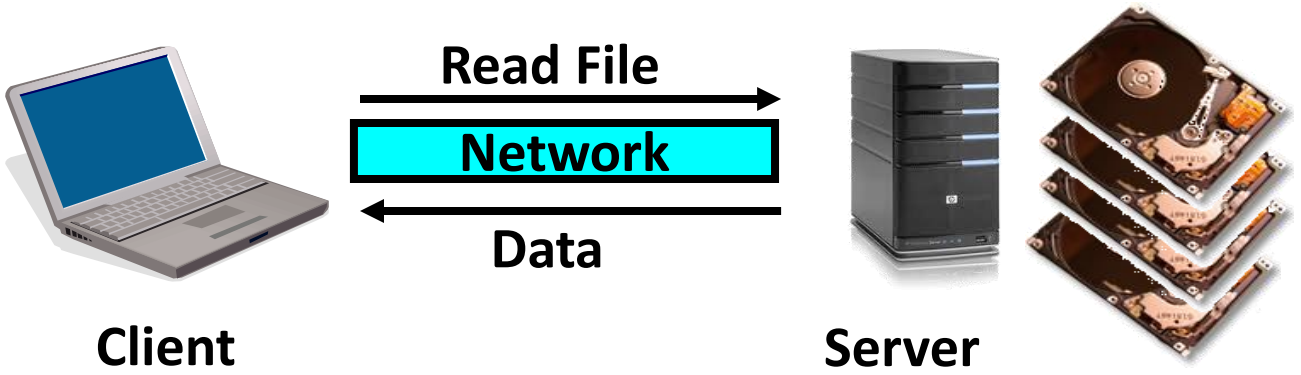
# Recall: Distributed Systems Motivation

- The *promise* of distributed systems
  - *Higher availability*: one machine goes down, use another
  - *Better durability*: store data in multiple locations
  - *More security*: each piece easier to make secure

- Other advantages too:
  - Cheaper/easier to build lots of simple computers
  - Allows for adding more resources incrementally
  - Users can have complete control over some components
  - Easier for users to collaborate

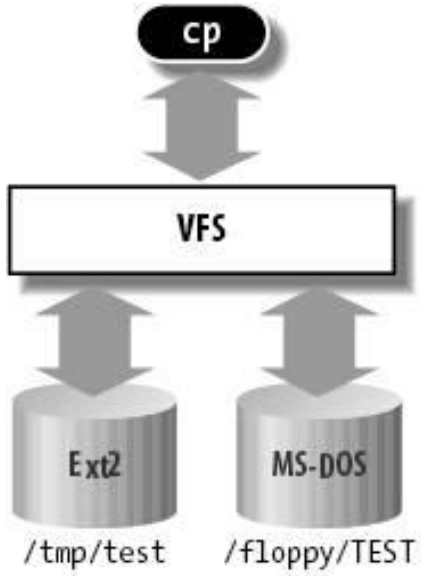
# Recall: Distributed File Systems

- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on campus file server



# Recall: Virtual Filesystem Switch (VFS)

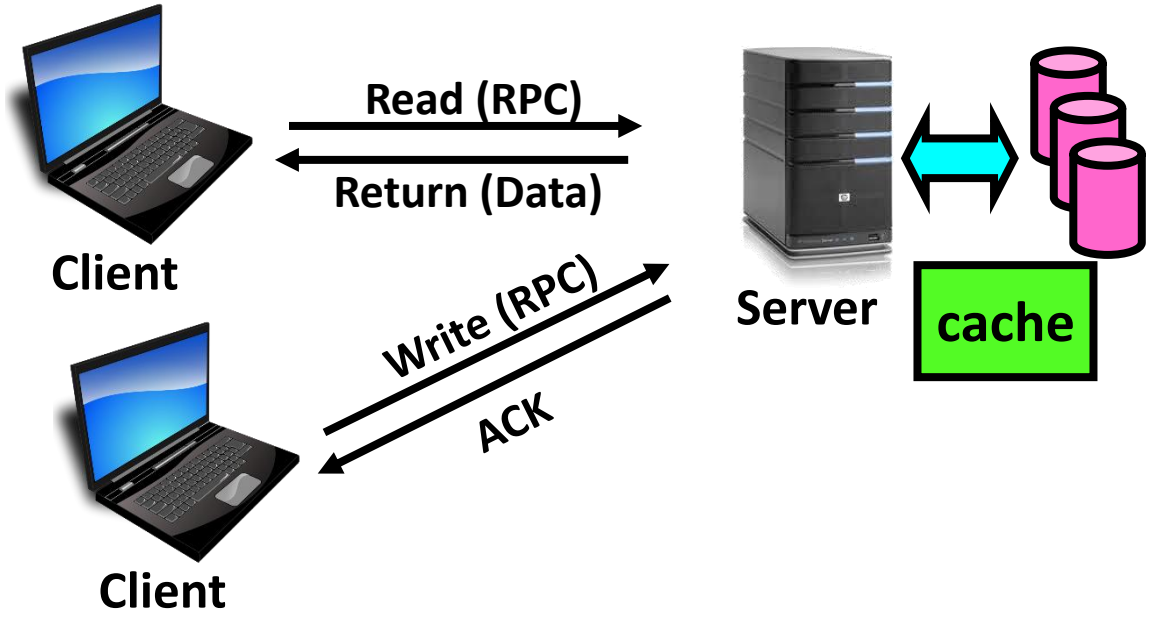
- Key idea: same system call interface is used to interact with many different types of file systems
  - **Dispatch** each system call to the appropriate filesystem-specific code
- Similar to device drivers: possible to plug in different implementations of the same interface



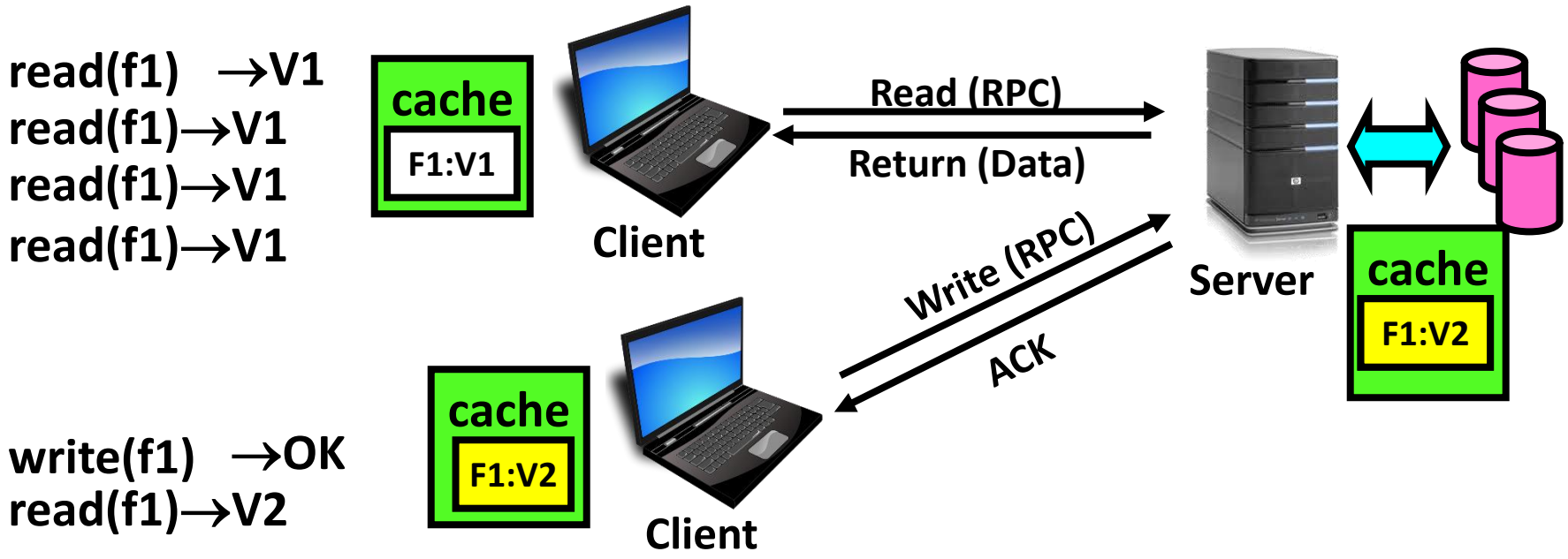

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

# Recall: Simple Distributed File System

- Remote Disk: Opens, Reads, Writes, Closes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - Server may cache files in memory to response more quickly
  - Server provides consistent view of file system to multiple clients
- Problem: performance (network slower than memory, server is bottleneck)

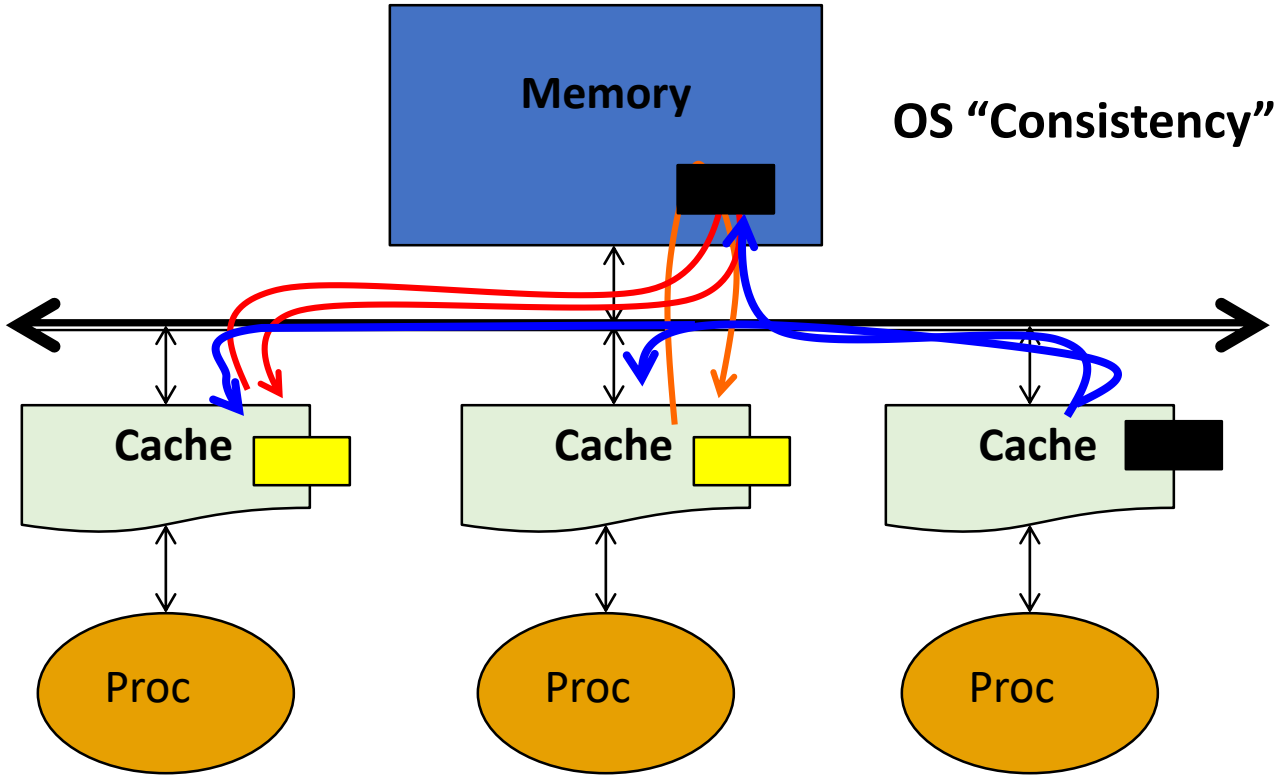


# Recall: Local Caching




- Idea: Use caching to reduce network load (e.g., buffer cache)
- New Problem: Consistency across caches

# Recall: Multiprocessor Cache Coherence




**OS “Consistency” ≈ Architecture “Coherence”**

- Interconnect is a broadcast medium (clients can observe all writes)
- Can’t use this strategy in a distributed file system!

# Recall: HTTP and State

- HTTP avoids this issue – *stateless protocol*
- Each request is self-contained
  - Treated independently of all other requests
  - Even previous requests from same client!

- So how do we get a *session*?
  - Client stores a unique ID locally – a ***cookie***
  - Client adds this to each request so server can customize its response

# Stateless Protocol

- A protocol in which all information required to service a request is included with the request
- Even better: Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)
- Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

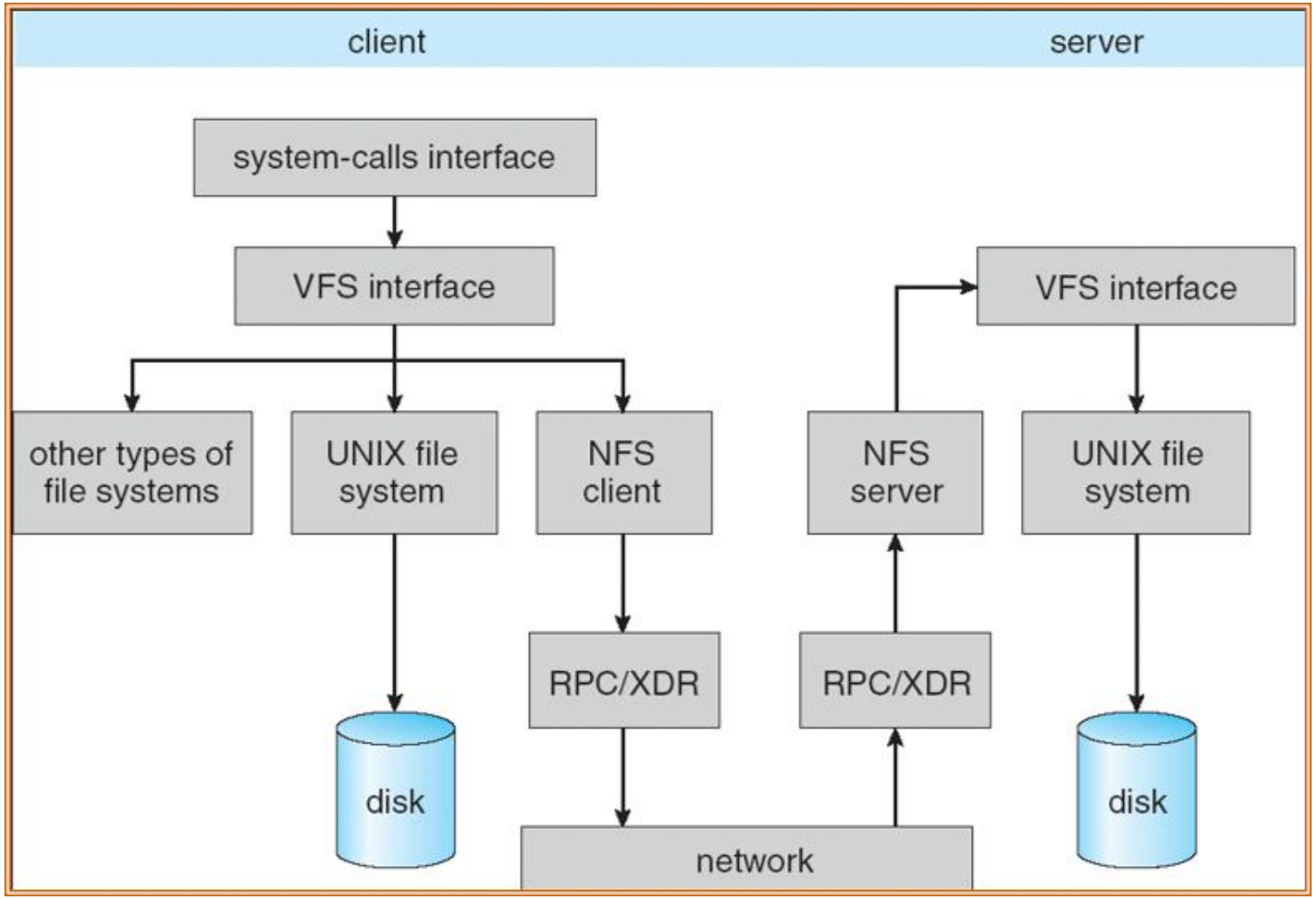
# NFS is Stateless

- NFS servers are stateless; each request provides all arguments required for execution
  - E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  - No need to perform network `open()` or `close()` on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - Hang until server comes back up (next week?)
    - Return an error. (Of course, most applications don't know they are talking over network)

# Network File System (Sun)

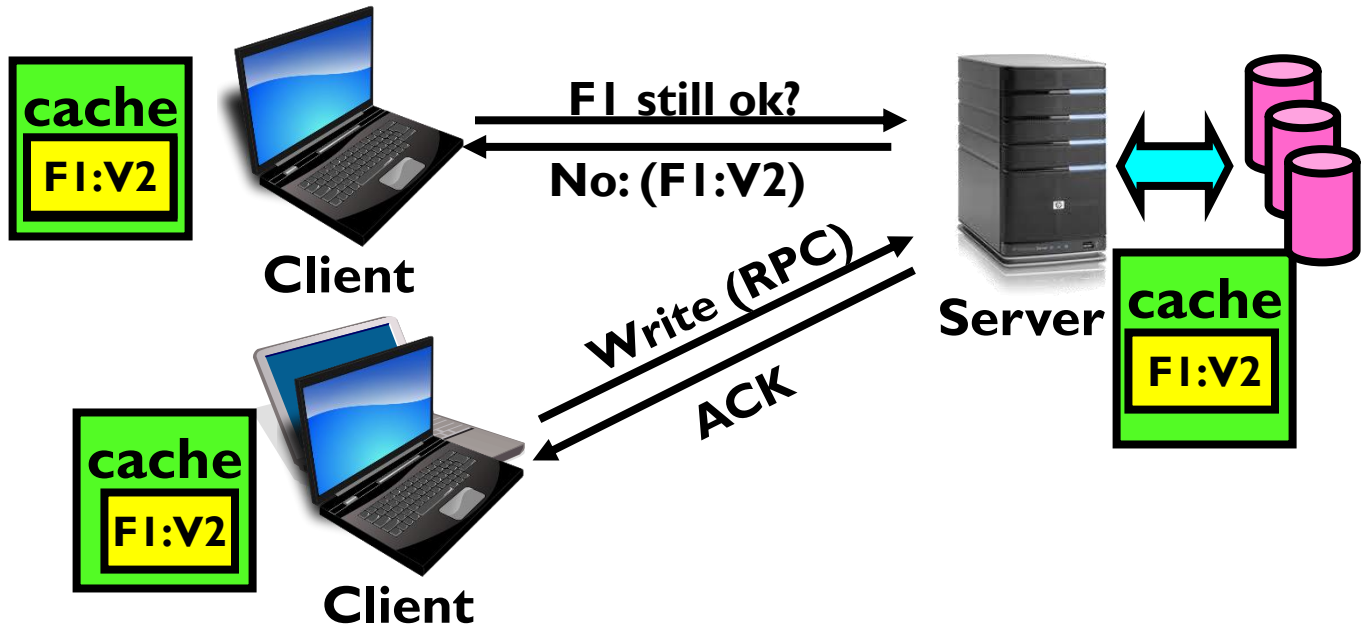
- Defines an RPC protocol for clients to interact with a file server
  - E.g., read/write files, traverse directories, …
  - Stateless to simplify failure cases
- Keeps most operations idempotent
  - Even removing a file: Return advisory error second time
- Don't buffer writes on server side cache
  - Reply with acknowledgement only when modifications reflected on disk

# NFS Architecture



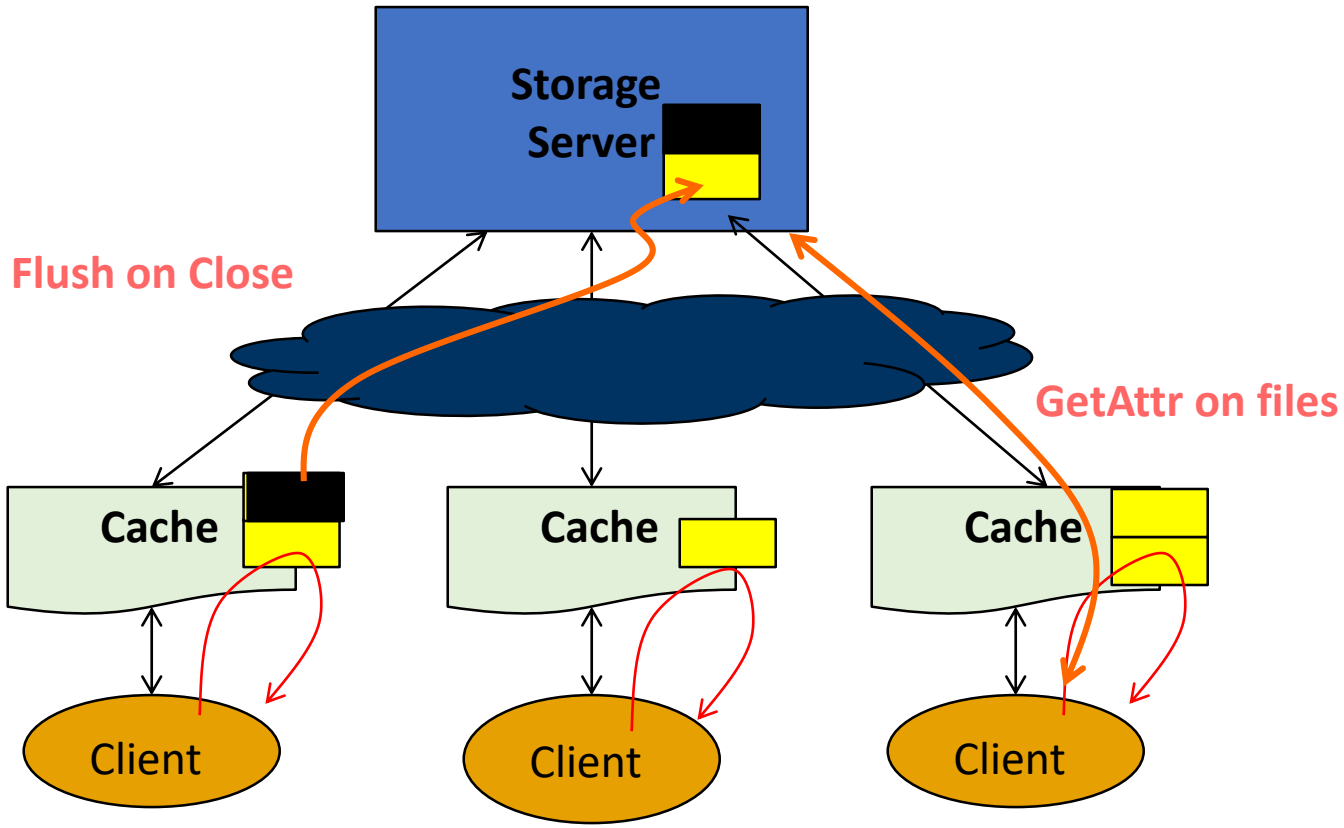
# NFS Cache Coherence

- Clients flush local changes to server on `close()`
- Clients periodically contact server to check if local file version is out of date
  - 3-30 sec. intervals (configuration parameter)




- What if multiple clients write to same file?
  - No guarantees: could see either version, or parts of both

# Cache Coherence in NFS: Summary




- Server allows multiple cached copies
- Update visibility by "flush on close" (and periodic flushing every 3 – 30 s)
- Clients check periodically for external modifications via GetAttr()

# Network File System: Pros/Cons

+ Simple, highly portable
  - Just need to speak RPC protocol to participate

- Sometimes inconsistent
- Doesn’t scale well to lots of clients
  - Clients keep checking to see if their caches stale
  - Server becomes bottleneck due to polling messages

# Moving Beyond NFS

- Stronger guarantees as to when you discover other clients' updates
  - Have the server inform the client of changes, instead of clients polling the server

- Some way of getting stronger guarantees as to when your data reaches the server (so others don't see partial updates)
  - Don't flush until close
  - Also reduces network traffic

# Andrew File System (AFS)

- Clients cache **entire files** (on local disk) rather than individual data blocks upon an open
- All reads/writes occur against local copy
  - Reduces network traffic
- Changes flushed to server on `close`
  - Clients don't see partial updates – all or nothing!
- *Callbacks* – **server** tracks who has copies of each file, **informs them** if their copy is now stale
  - Client will fetch new version on next open

# Andrew File System (AFS)

- Clients no longer need to poll server for cache invalidation, less network traffic

- Client disk as cache: More files can be cached
  - Read only workload: No need to involve server

- Consistency still has issues, but easier to describe
  - Two clients have file open at same time and both write: last to `close` wins (overwrites other client's update)

# Failure in AFS

- Client fails?
  - Need to double check validity of all cached files
  - May have missed callback alerts from server while down
- Server fails?
  - Clients must be made aware of this
  - Clients must reestablish callbacks
- Callbacks mean server maintains more state than in NFS design

# Andrew File System: Pros/Cons

+ Less server load than NFS
  - Disk as cache: clients can cache more files locally
  - Callbacks: server is not involved for read-only files
+ Easier to reason about consistency model
  - Clients don't see partial updates

- Server maintains more state
- Inefficient to download whole file on open
  - Especially if only part of it is used

# NFS/AFS Issues

- Performance: Central file server is a bottleneck
- Availability: Server is a single point of failure
- Higher cost for server hardware, maintenance compared to client machines

# Announcements

- Homework 5 is due on Friday

- Work on Project 3
- After design reviews are over…
  - Will release a sample design doc after all design reviews are over
  - We might also simplify the synchronization requirement

# Distributed Key-Value Stores

# Key-Value Stores (KV Store)

- Simple Interface:
  - `put(key, value); // Insert/write value associated with key`
  - `get(key); // Retrieve/read value associated with key`

- Behaves just like a dictionary in Python

- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

# Why Key-Value Storage?

- Simplicity → Easy to Scale
  - Uniform items: distribute easily and roughly equally across many machines
  - Handle huge volumes of data (e.g., petabytes)

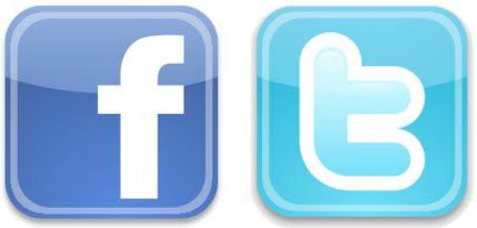
- Relatively simple consistency properties
  - Simpler than general database transactions
  - Simple than file system ops over multiple blocks

- Building block for more capable databases…

# Key-Value Stores: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)


- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)



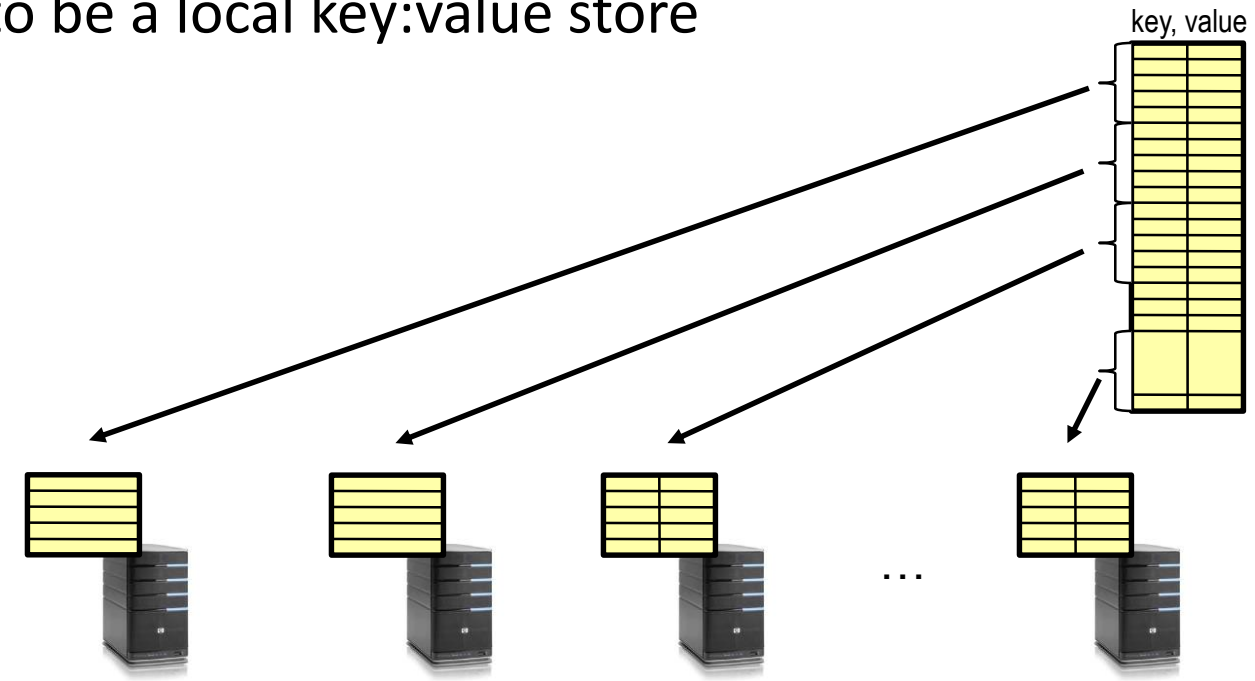
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

# Key-Value Storage Systems in the Wild

- **Amazon**
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
  - All the different services share distributed systems infrastructure
- **Cassandra**: "distributed data management system" (developed by Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)

# Key-Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-value pairs across many machines
  - Easy to be a local key:value store



# Distributed Key-Value Store vs. Distributed File System

- Distributed File System
  - One server
  - Challenge: keep clients' caches consistent

- Distributed Key-Value Store
  - Many servers
  - Challenge: keep server state consistent
  - Challenge: take advantage of multiple servers to scale the system
  - (Often, safe to assume no cache at the client --- or if it's there, it isn't part of the consistency problem)

# Important Questions

- **put(key, value)**:
  - where do you store a new (key, value) tuple?
- **get(key)**:
  - where is the value associated with a given “key” stored?

- And, do the above while providing :
  - Fault Tolerance: gracefully handle machine failures
  - Scalability: scale to many machines, easy to add new machines
  - Consistency: keep data consistent despite failures/message losses

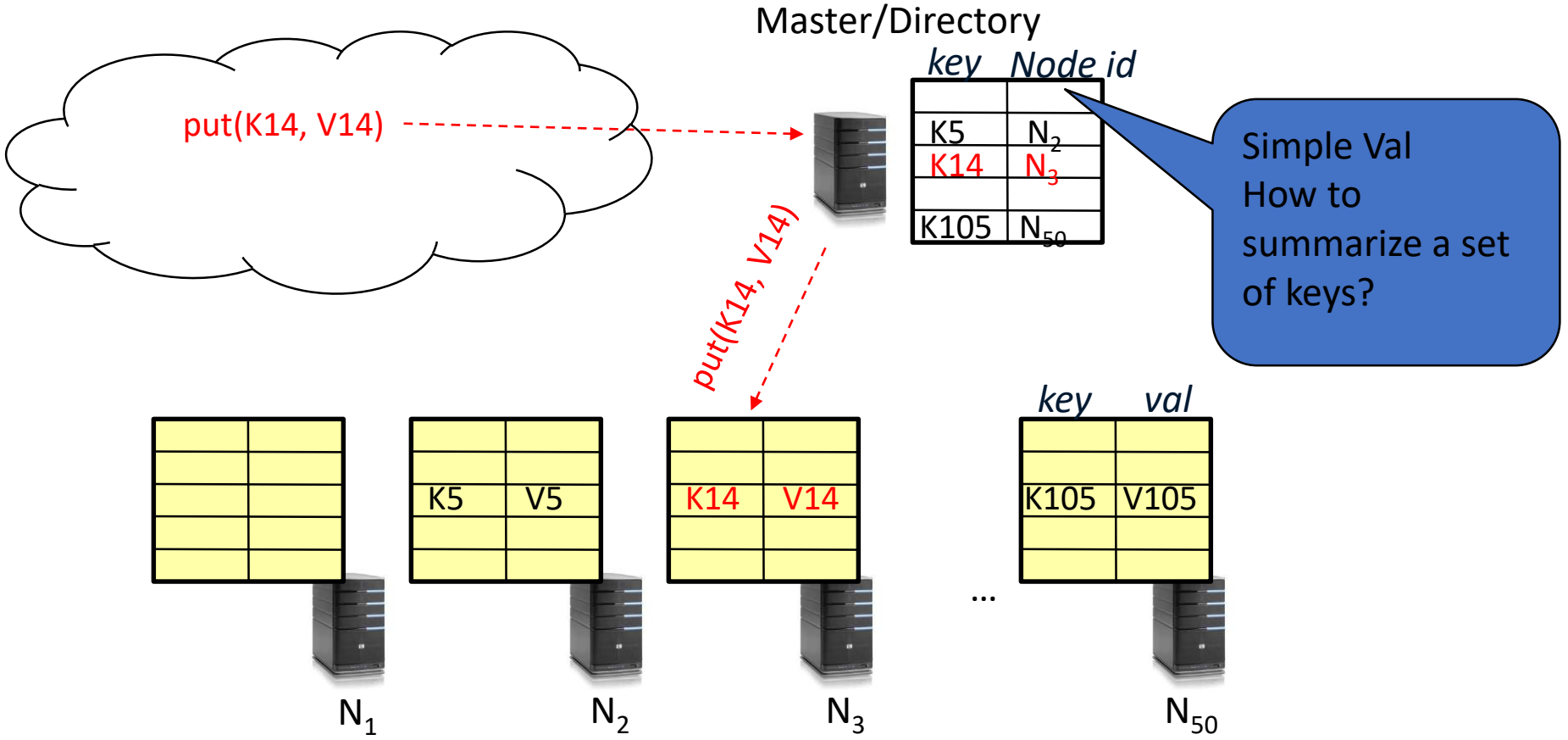
# How to Solve the "where"?

- Use hash function to map each key to a location
  - Key:Value is stored on the node whose ID is Hash(Key)
  - But… what if you don't know who are all the nodes that are participating?
  - Perhaps they come and go…
  - What if some keys are really popular?

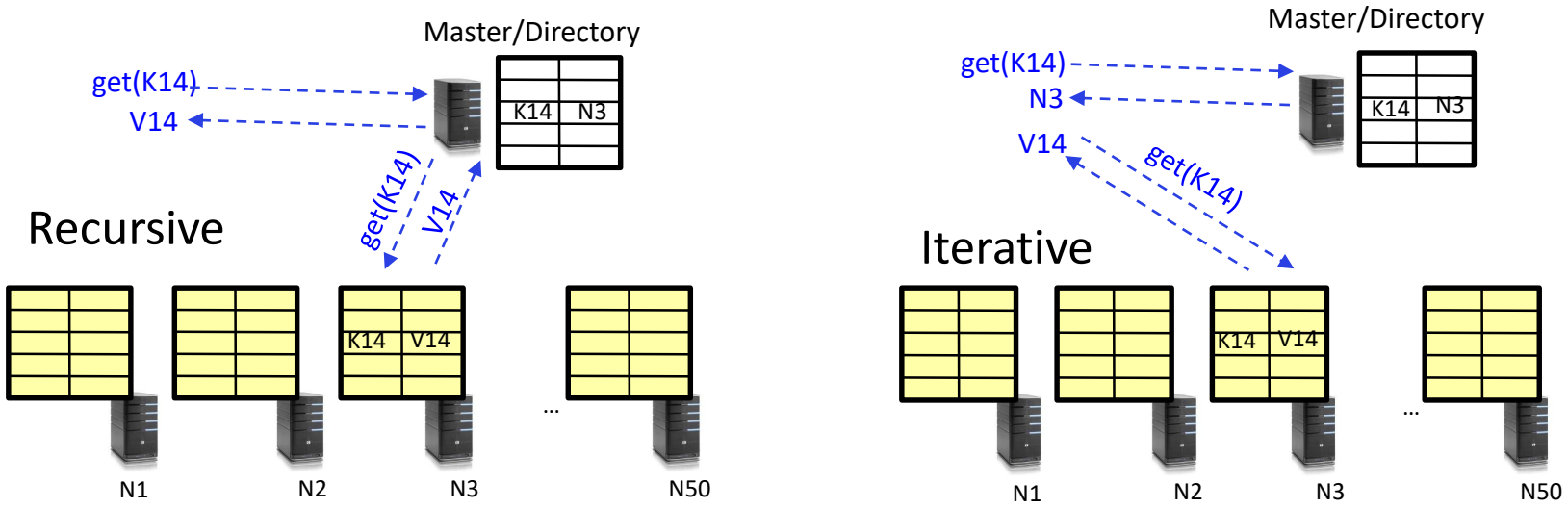
- Look it up in a directory
  - The directory is just another key-value store
  - But it can be *centralized* (since locations are often smaller than values)

# Directory-Based Architecture

- Have a node maintain the mapping between **keys** and the ***machines* (nodes)** that store the **values** associated with the **keys**



# Iterative vs. Recursive Query




- Recursive Query: Directory server queries the appropriate node
- Iterative Query: Directory server tells client the node; then client queries that node

# Iterative vs. Recursive Query

**Recursive**

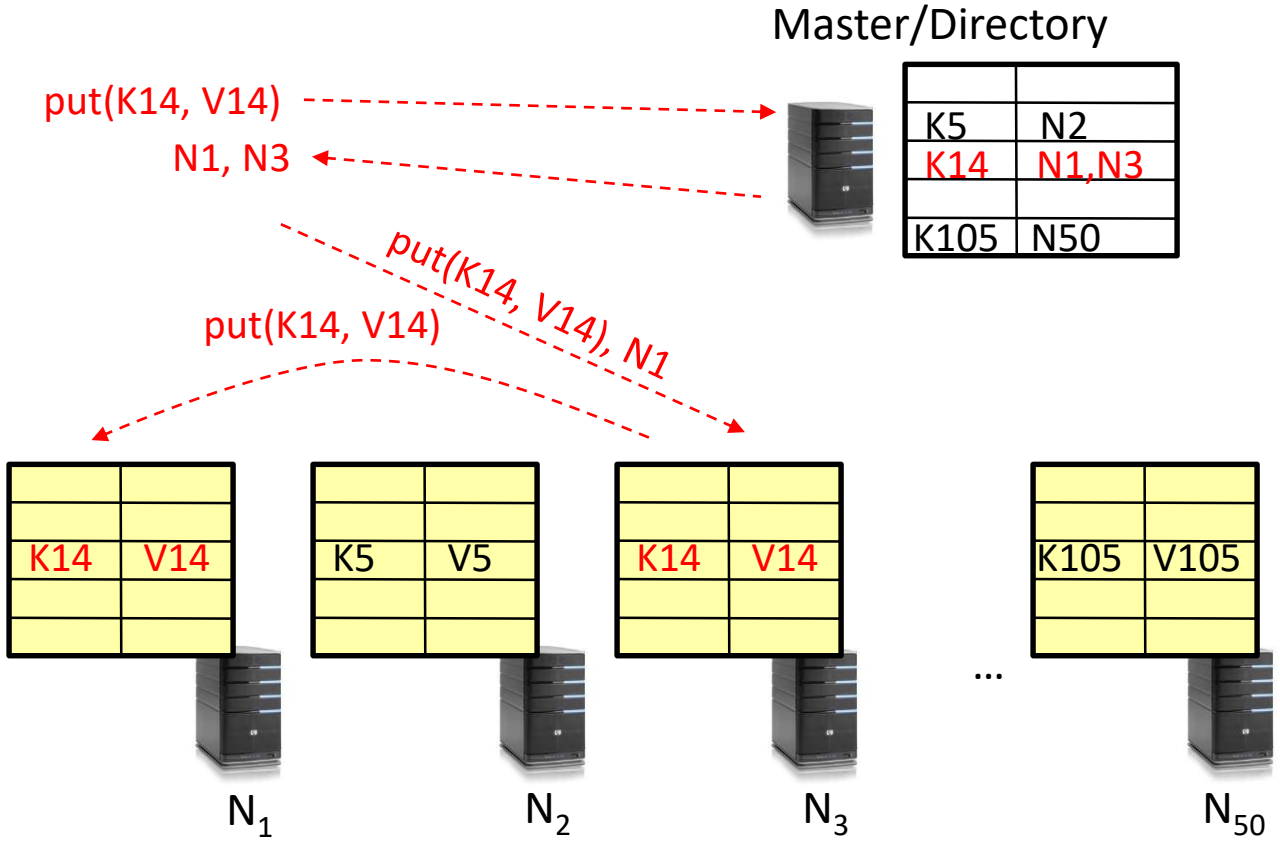
+ Faster, as directory server is typically close to storage nodes
+ Easier for consistency: directory can enforce *an order* for all puts and gets
- Directory is a performance bottleneck

**Iterative**

+ More scalable, clients do more work
- Harder to enforce consistency

# Directory-Based KV Store: Fault Tolerance

- **Replicate** value on several nodes
- And try to place replicas on different racks (to avoid correlated failures)

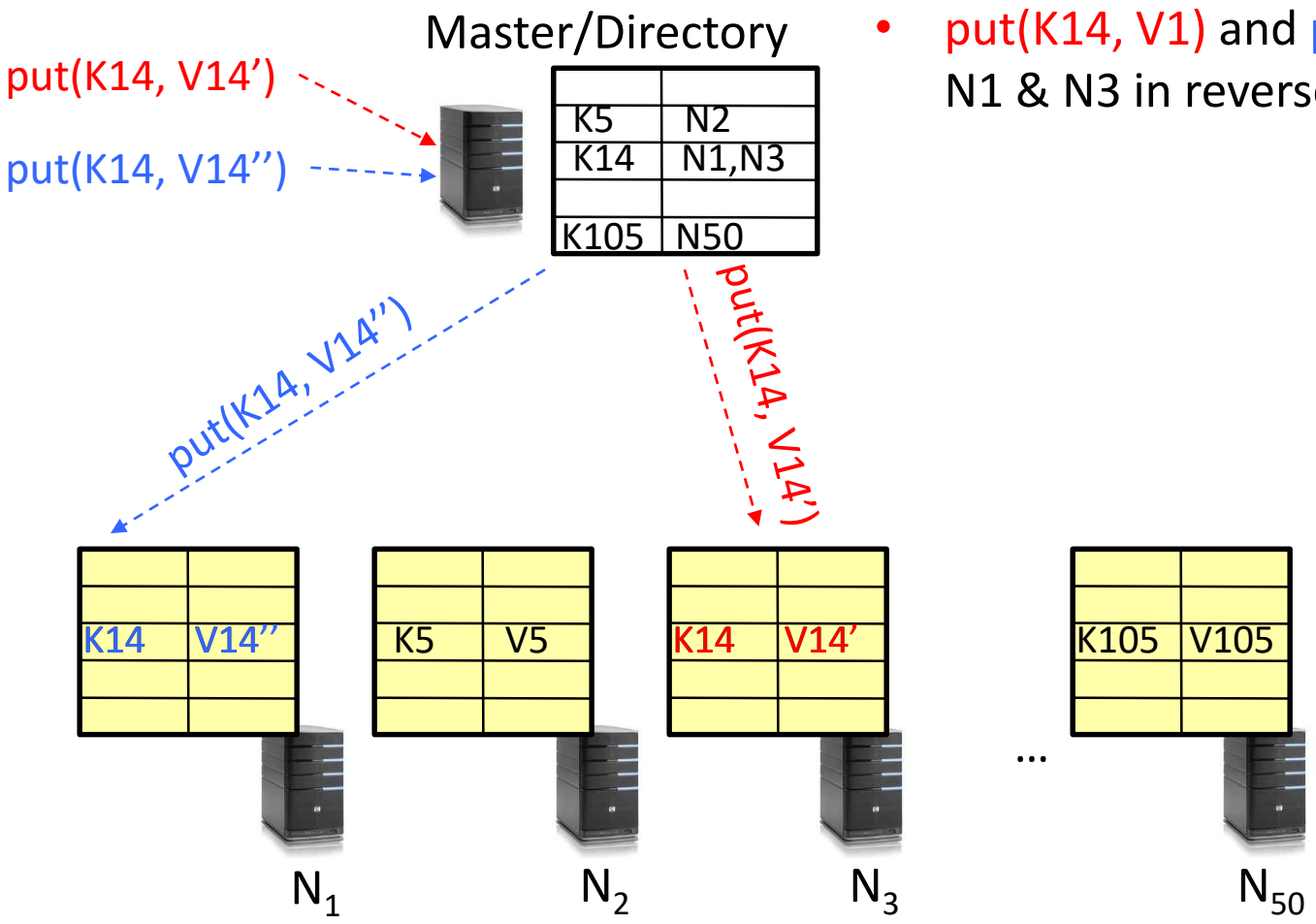


# Consistency

- Need to make sure a value is replicated correctly
- How do you know a value is replicated on every expected node?
- *Wait* for acknowledgments from all expected nodes???

# Directory-Based KV Store: Consistency

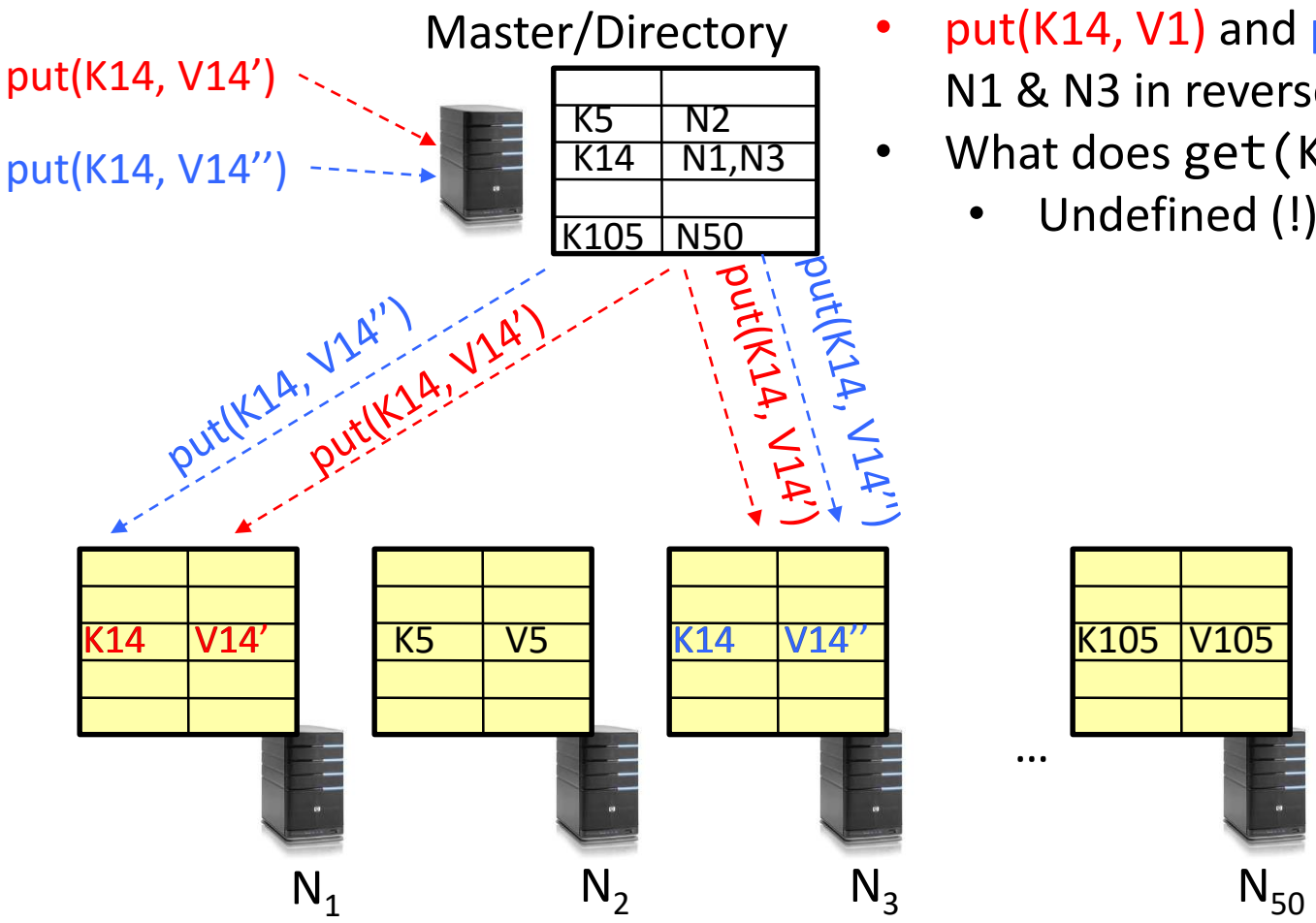
- For concurrent updates, need to make sure that updates to a key are **applied to the replicas in the same order**




- put(K14, V1) and put(K14, V2) reach N1 & N3 in reverse order

# Directory-Based KV Store: Consistency

- For concurrent updates, need to make sure that updates to a key are **applied to the replicas in the same order**




- put(K14, V1) and put(K14, V2) reach N1 & N3 in reverse order
- What does `get(K14)` return?
  - Undefined (!)

# Reliability, Performance, and Consistency

- What is the key to **reliability** and **performance**?

**Replication**

- What is the source of **inconsistency**?

**Replication**

# How to ensure that PUTs are applied to the replicas in order?

- **Wait** for explicit acknowledgment
  - Just like synchronization to order threads in a process…

- But doesn't this make it run slower???
  - Intuitively, the system should run *faster*, since we have more replicas to work with (remember RAID?)

# Consistency and Fault Tolerance

- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down entire `put`? Pick another node?
- In general with multiple replicas: slow `put` and fast `get` operations
- But it turns out that this isn't fundamental…

# Quorum Consensus

- Improve `put` and `get` operation performance **in the presence of replication**
  - Reduce the number of replicas to hear back from

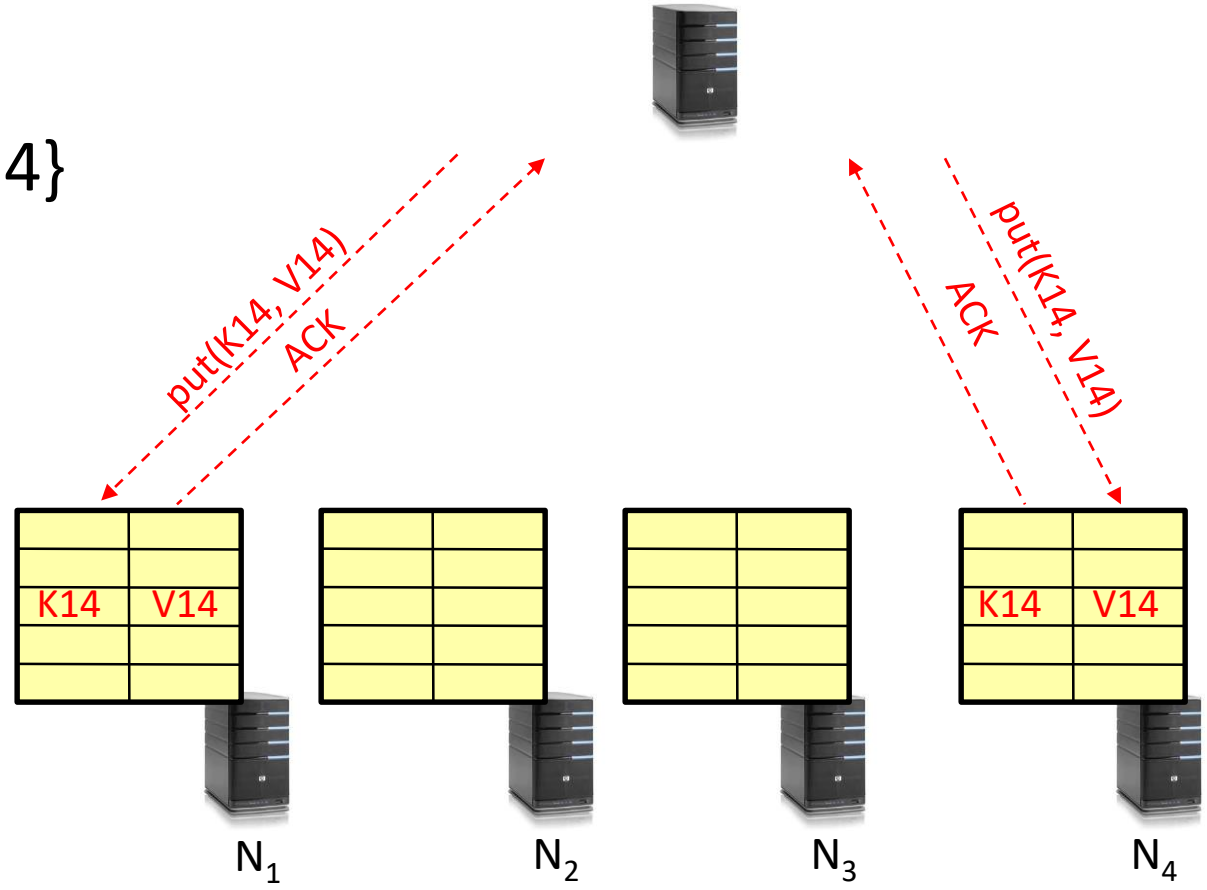
- Define a replica set of size N
  - `put` waits for acknowledgements from at least ***W*** replicas (a "write quorum")
  - `get` waits for responses from at least ***R*** replicas (a "read quorum")
  - Make sure that ***W + R > N***

- Why does it work?
  - If ***W + R > N***, then every read quorum has at least one replica in common with the write quorum used for the latest write

- Why might you use W+R > N+1?

# Quorum Consensus Example

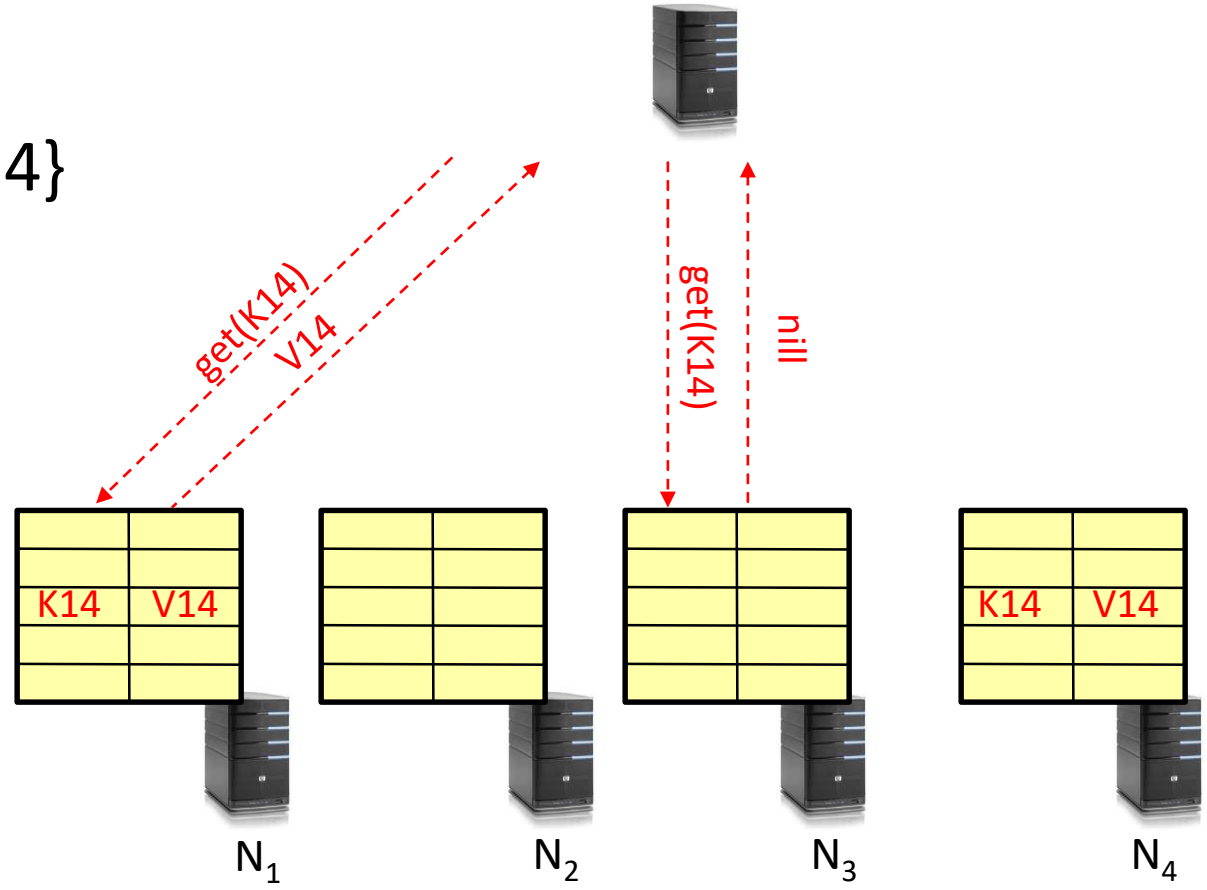
- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}



# Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}

- Issuing `get` to any two out of three will return the answer



# Scalability

- **How easy is it to make the system bigger?**

- Scaling Amount of Storage: Use more nodes
- Scaling Number of Requests
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular item on more nodes
- Scaling the Master/Directory Node
  - Replicate It (multiple identical copies)
    - Maintain consistency across them
  - Partition it, so different keys are served by different directories

# Scalability: Load Balancing

- Directory tracks available storage at each node
  - Prefer to insert at nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values at new node
  - Move values from heavily loaded nodes to new node
- What happens when a node fails?
  - Replicate values from failed node to other nodes

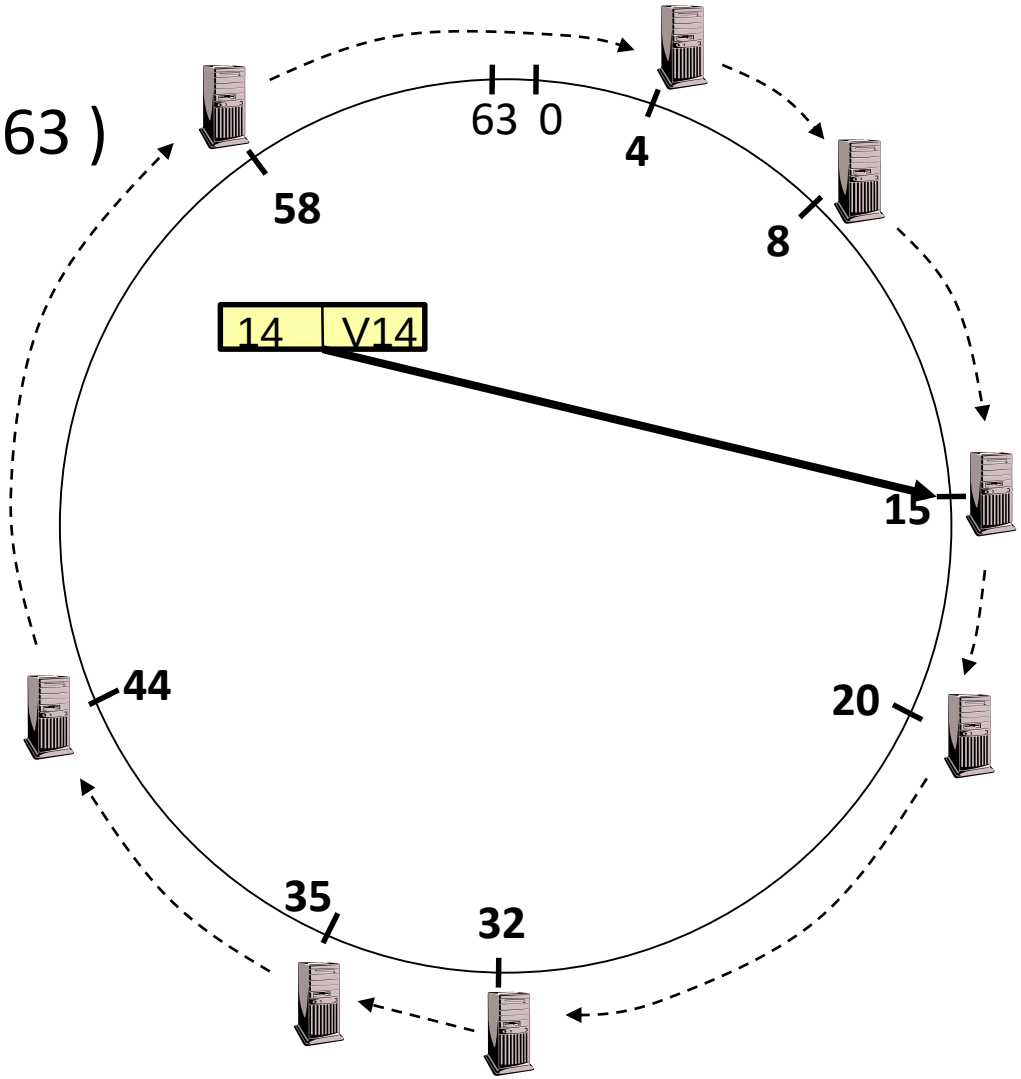
# Scaling the Directory

- Directory contains number of entries equal to number of key/value pairs in entire system
  - Could be tens or hundreds of billions of pairs
- Solution: **Consistent Hashing**
  - The set of storage nodes may change dynamically
    - fail, enter, leave
  - Assign each node a unique ID in large namespace [0..$2^m$-1]
    - m bit namespace, s.r., M << $2^m$
    - Each node can pick its ID at random !
  - hash keys in a manner that everyone assigns same range of IDs to a node
  - Each (key,value) stored at node with *smallest ID larger than hash(key)*
- Important property: Adding a new bucket doesn't require moving lots of existing values to new buckets

# Key-to-Node Mapping Example

Partitioning example with m = 6 (ID space: 0..63 )
0: Node 4 maps keys [59, 4]
1: Node 8 maps keys [5,8]
2: Node 15 maps keys [9,15]
3: Node 20 maps keys [16, 20]
…
M-1: Node 58 maps [45, 58]

n := Hash(key)
Find first *i* in [4, 8, 15, 20, …]
s.t., $N_i > n$ (mod M)



# Scaling the Directory even further…

- For the directory to hash a key to a node, it must know the IDs of ***all*** nodes in the system
  - Still a central point of failure!

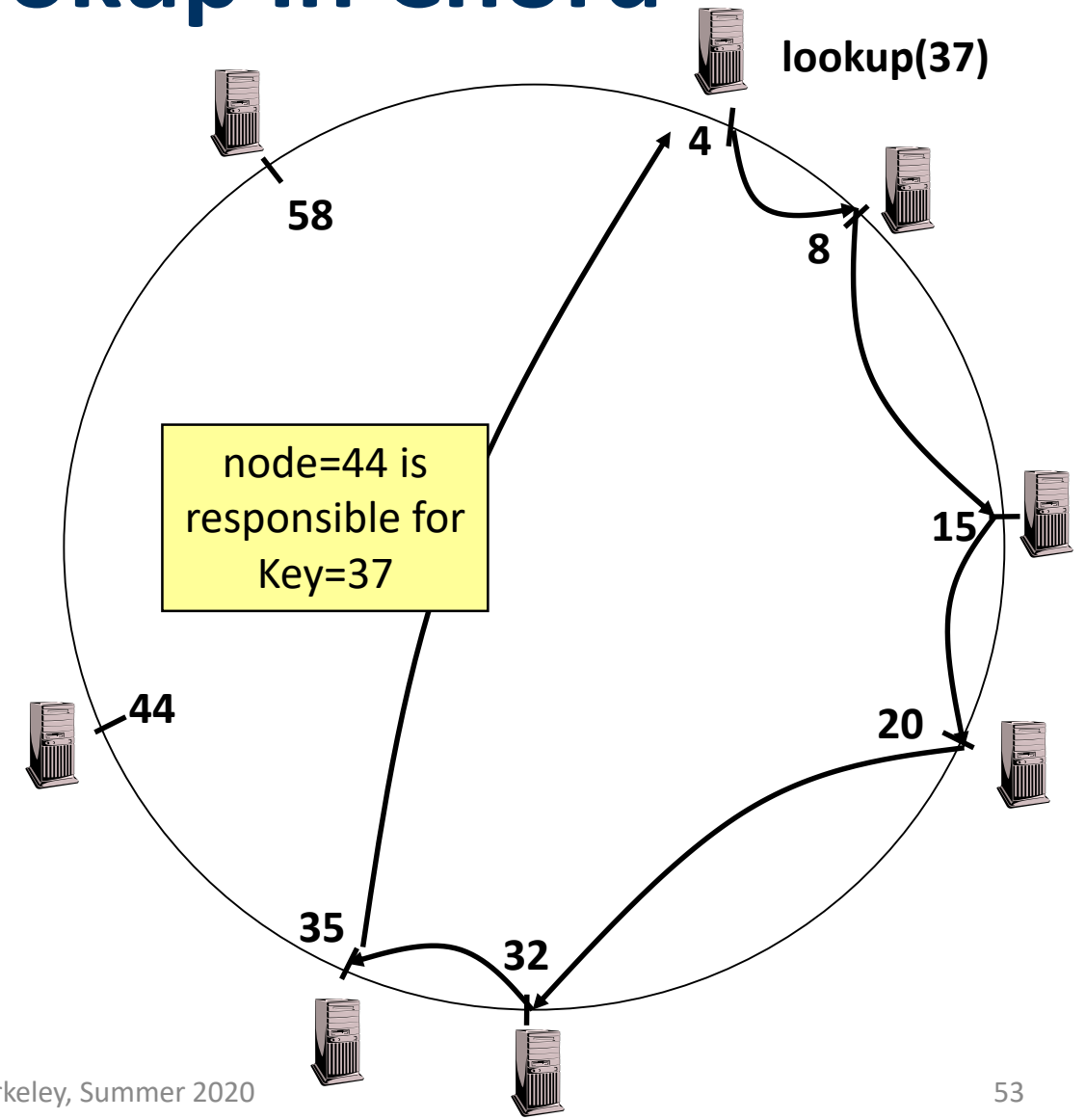
- Can we design a distributed KV store where no node has to know about all the other nodes? Or even most of them?
  - Yes! The Chord DHT does this!
  - For N nodes, each node keeps track of log(N) others
  - Can look up a key in log(N) network round-trips
  - Directory is distributed among all nodes

# Chord: Distributed Directory Service

- Import aspect of the design space:
  - Decouple correctness from efficiency
  - Combined *Directory* and *Storage*
- Properties
  - Correctness:
    - Each node needs to know about neighbors on ring (one predecessor and one successor)
    - Connected rings will perform their task correctly
  - Performance:
    - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
    - Guarantees that a tuple is found in O(log(*M*)) steps
- Many other *Structured, Peer-to-Peer* lookup services:
  - CAN, Tapestry, Pastry, Bamboo, Kademlia, …
  - Several designed here at Berkeley!

# Basic (Unoptimized) Lookup in Chord

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
  - E.g., node=4 lookups for node responsible for Key=37
- Worst-case (correct) lookup is O(n)
  - But with optimizations (next slide), we can achieve an O(log n) lookups in the normal case

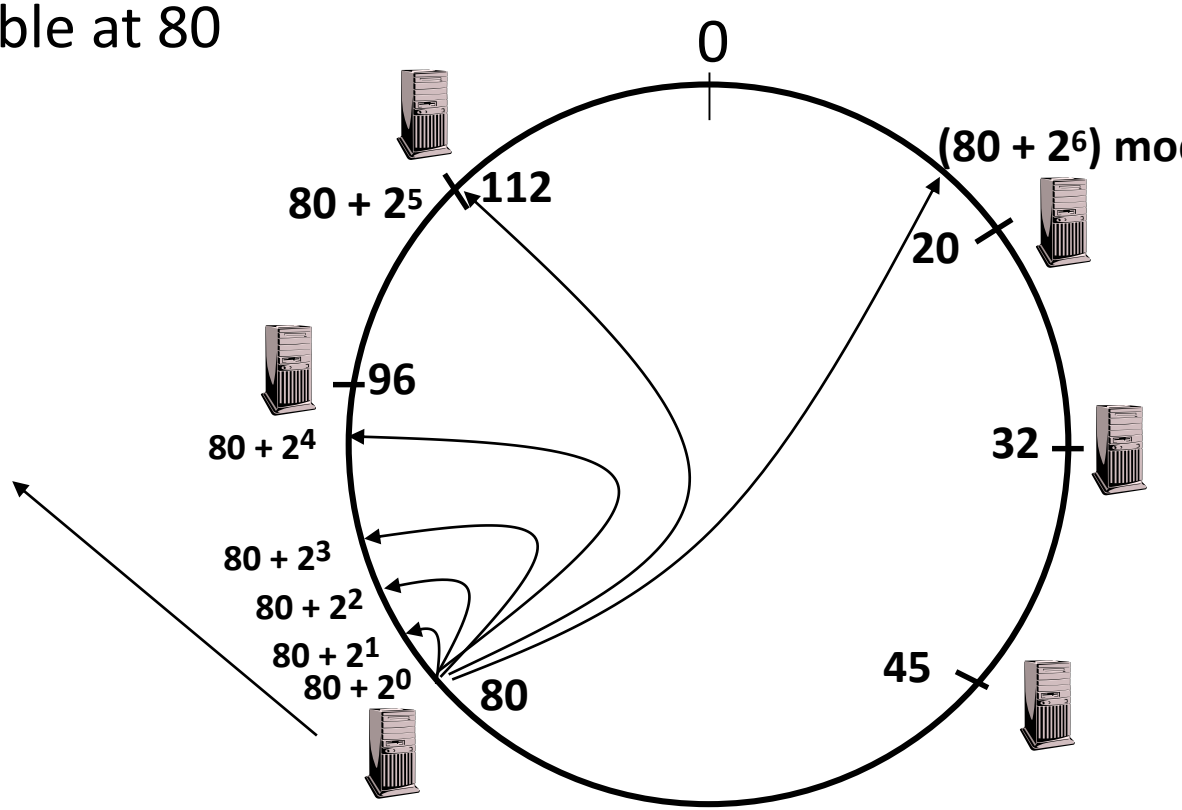


# Efficient Lookup in Chord: Finger Tables

Say *m=7*

Finger Table at 80

| *i* | *ft[i]* |
|---|---|
| 0 | 96 |
| 1 | 96 |
| 2 | 96 |
| 3 | 96 |
| 4 | 96 |
| 5 | 112 |
| 6 | 20 |




**"Short-circuit" the lookup by following the fingers (chords) in the circle where possible**

$i$th entry at peer with id $n$ is first peer with id >= $n + 2^i \pmod{2^m}$

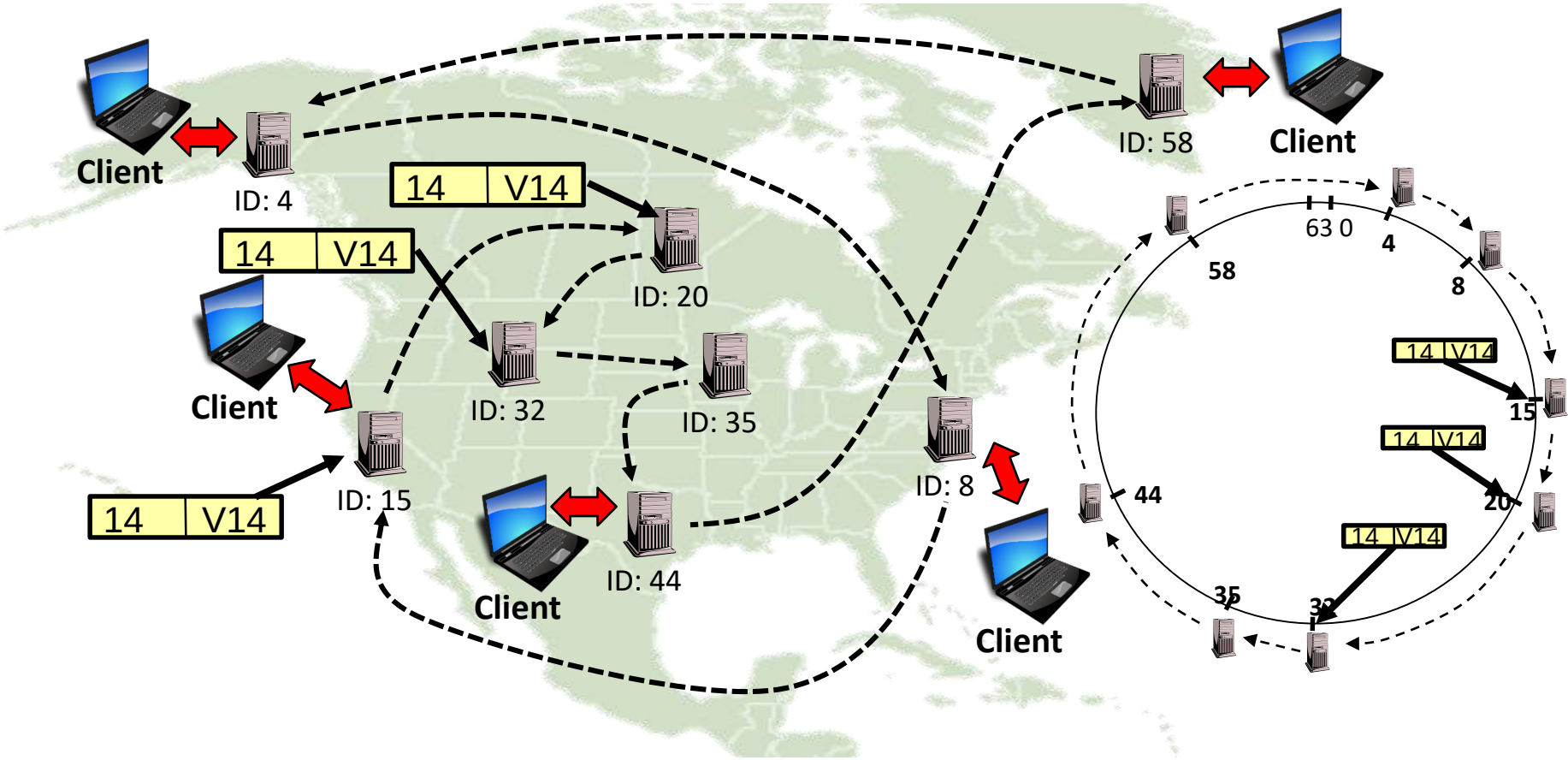
# Various Other Measures in Chord

- Robustness in the face of fault-tolerance
  - Remember not just the successor, but the next $k$ successors
  - Store key:value not just on the node that results from lookup, but also the next few successors…

- What if a new node joins?
  - Use directory mechanism to "look up" its place on the ring and find its predecessor and successor
  - Coordinate with its new successor and predecessor to notify them of its presence
  - Obtain a copy of the data it's supposed to hold

# Replication in Physical Space

- Replicating in adjacent nodes of virtual space → geographic separation in physical space
  - Avoids single points of failure through randomness



# Summary

- Key Value Store: Simple `put` and `get` operations
  - Fault tolerance: replication
  - Scalability: Add nodes, balance load, no central directory
  - Consistency: Ordered updates
    - Quorum consensus for better performance
- Chord:
  - Highly scalable distributed lookup protocol
  - Each node needs to know about O(log(M)), where m is the total number of nodes
  - Guarantees that a tuple is found in O(log(M)) steps
  - Highly resilient: works with high probability even if half of nodes fail