# CS162
# Operating Systems and Systems Programming
# Lecture 23

## Distributed Storage, Key-Value Stores

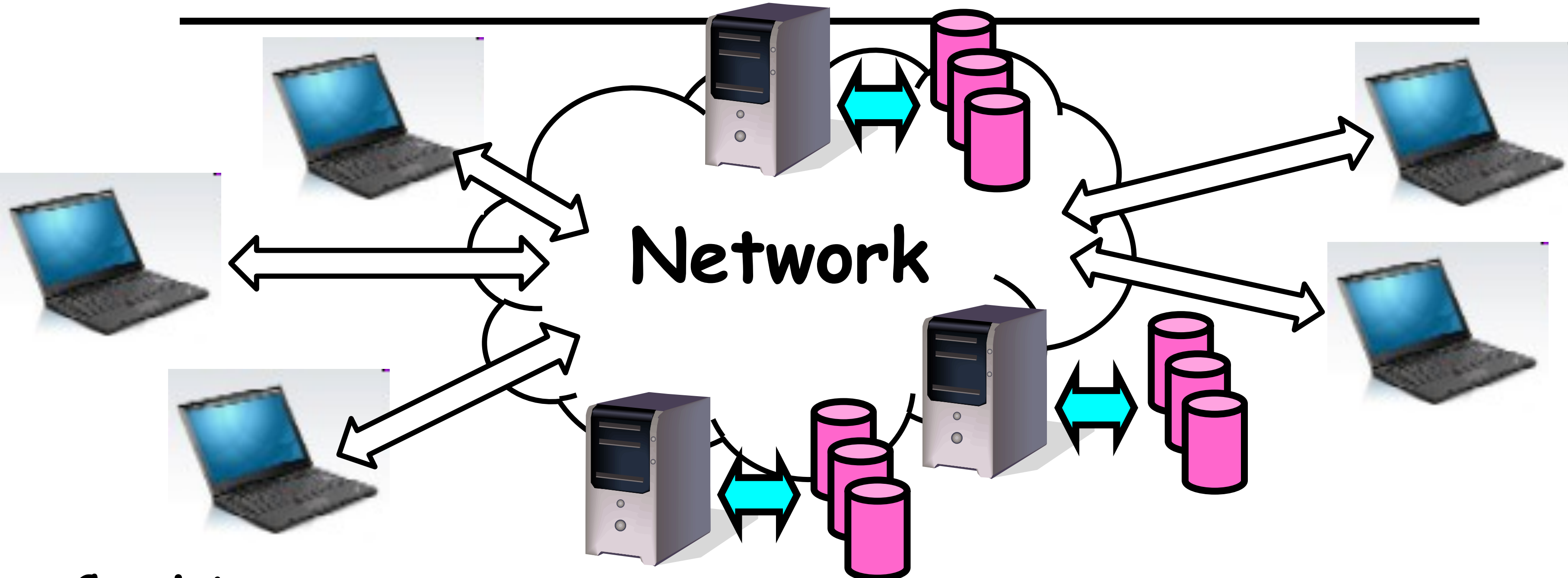
**November 30th, 2015**

**Prof. John Kubiatowicz**

**http://cs162.eecs.Berkeley.edu**

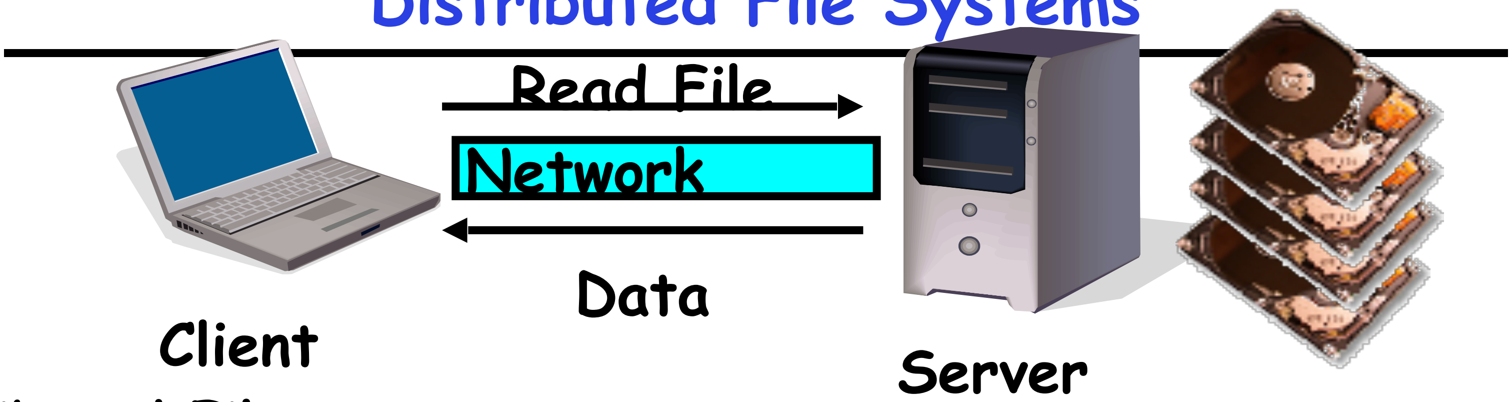
*Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at* Berkeley, with few minor updates/changes*. When slides are obtained from other sources, a  a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

# Network-Attached Storage and the CAP Theorem

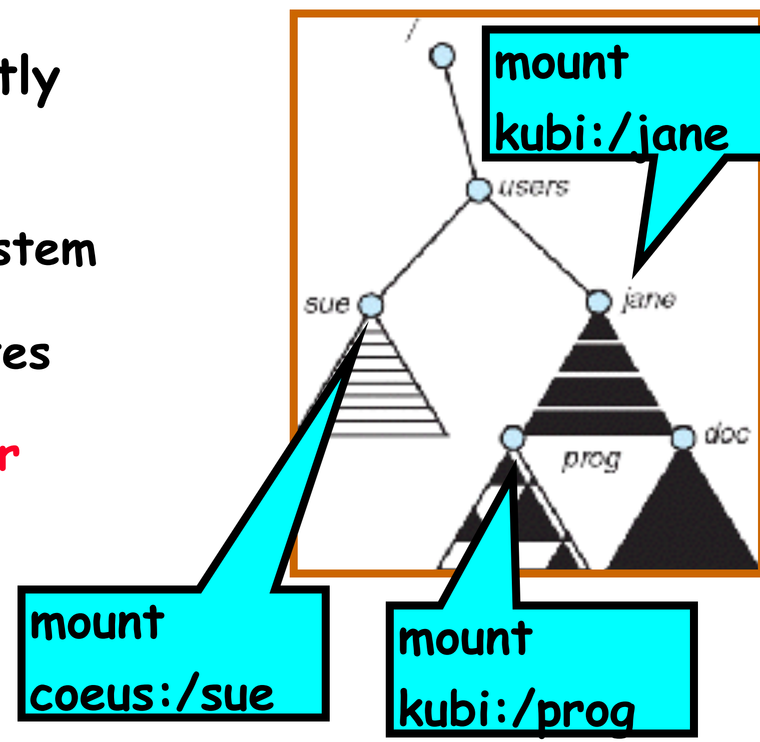



- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as "Brewer's Theorem"

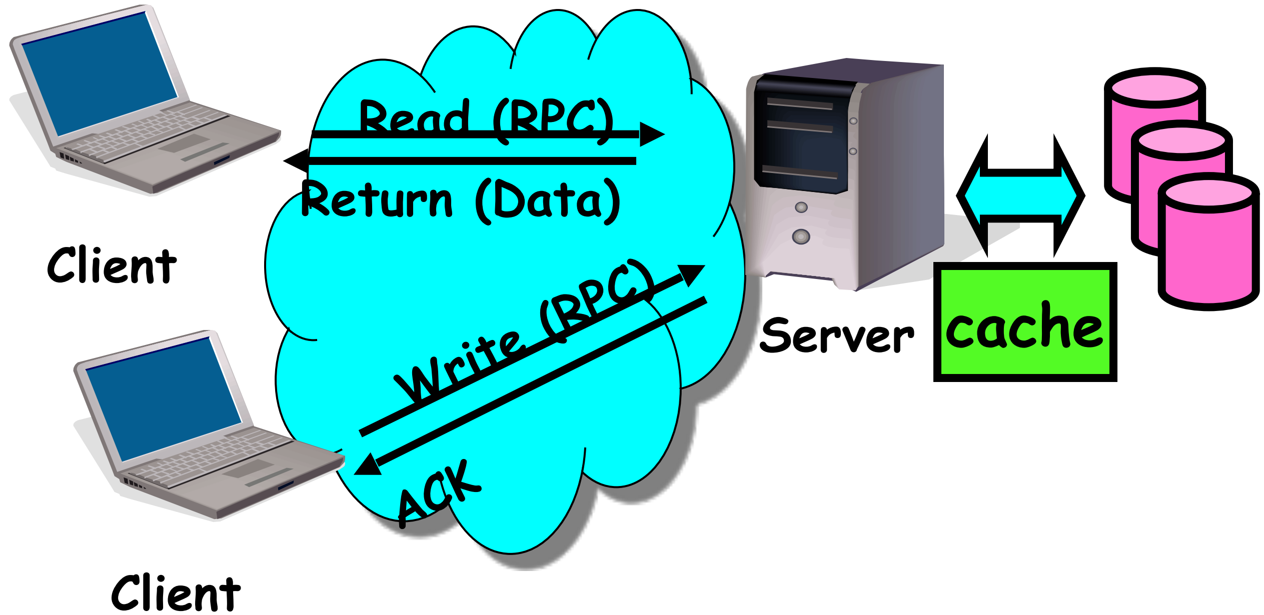
# Distributed File Systems




- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - Hostname:localname: Name files explicitly
    - » No location or migration transparency
  - Mounting of remote file systems
    - » System manager mounts remote file system by giving name and local mount point
    - » Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - A single, global name space: every file in the world has unique name
    - » Location Transparency: servers can change and files can move without involving user

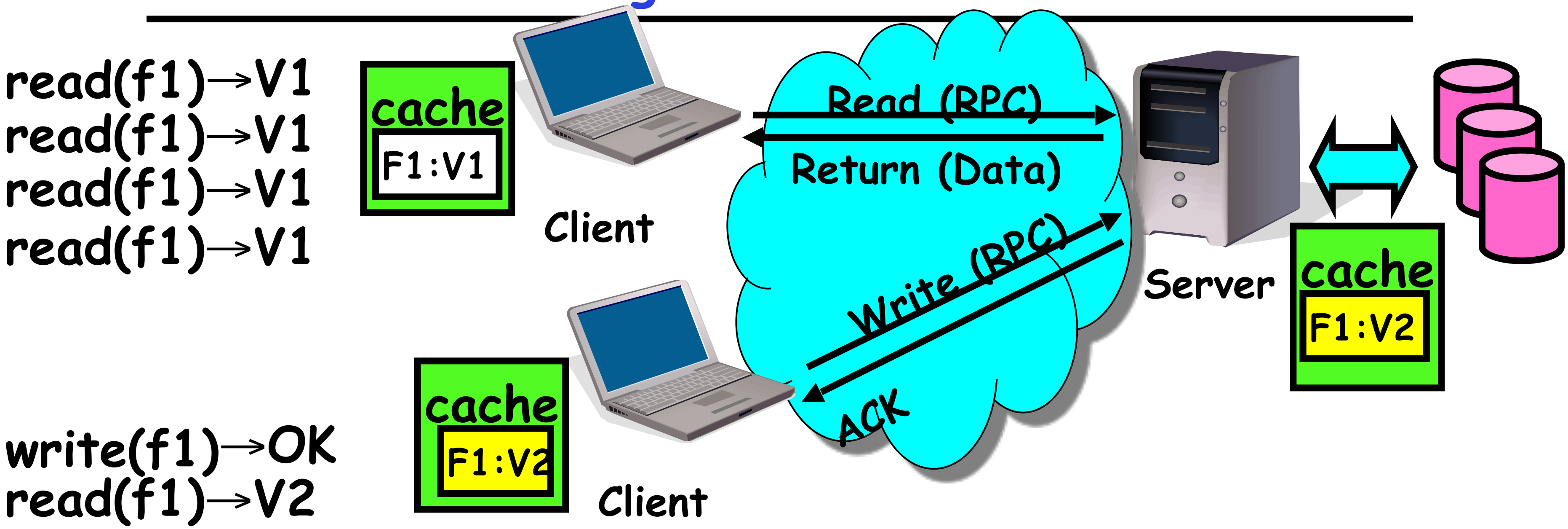


# Simple Distributed File System



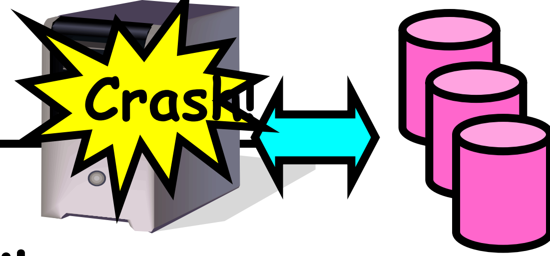

- Remote Disk: Reads and writes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

# Use of caching to reduce network load




- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

# Failures




- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "rm foo", but before acknowledgment?
    - » Message system will retry: send it again
    - » How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- Stateless protocol: A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

# Network File System (NFS)

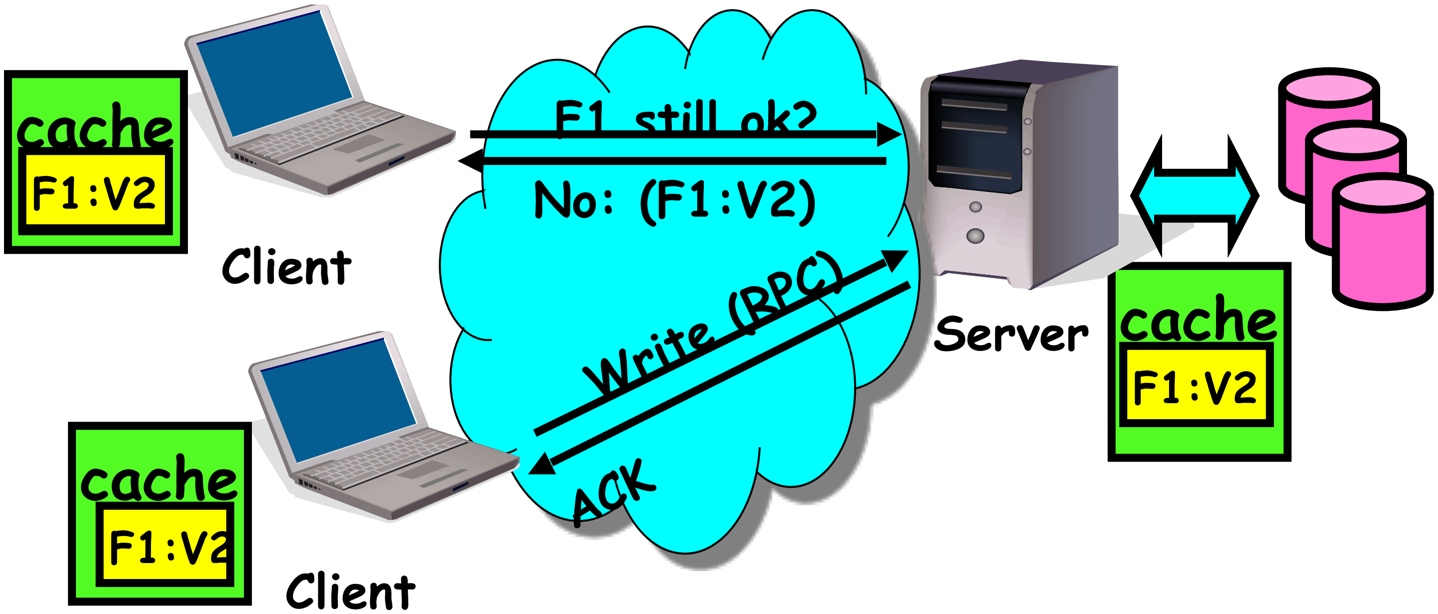
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

# NFS Continued

- NFS servers are stateless; each request provides all arguments require 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)

# NFS Cache consistency

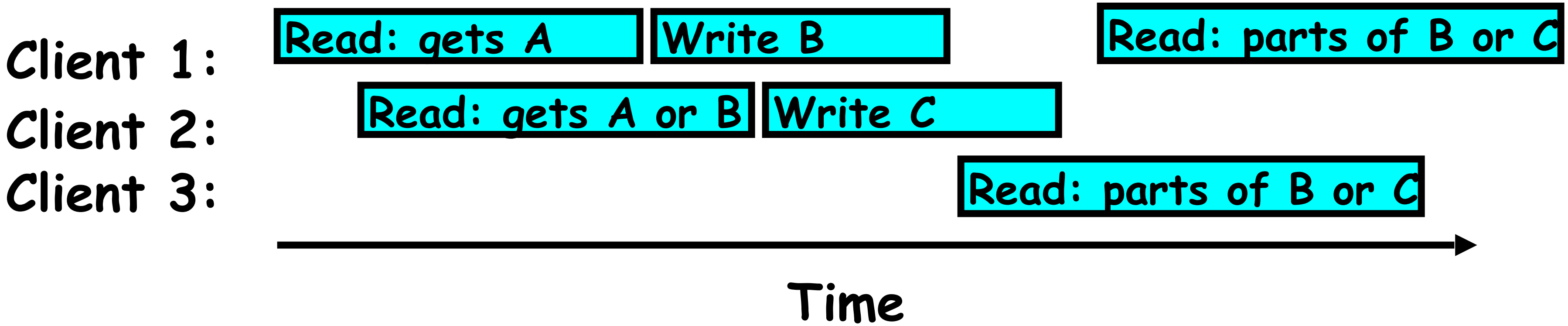
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.




  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

# Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"




- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

# NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large # clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

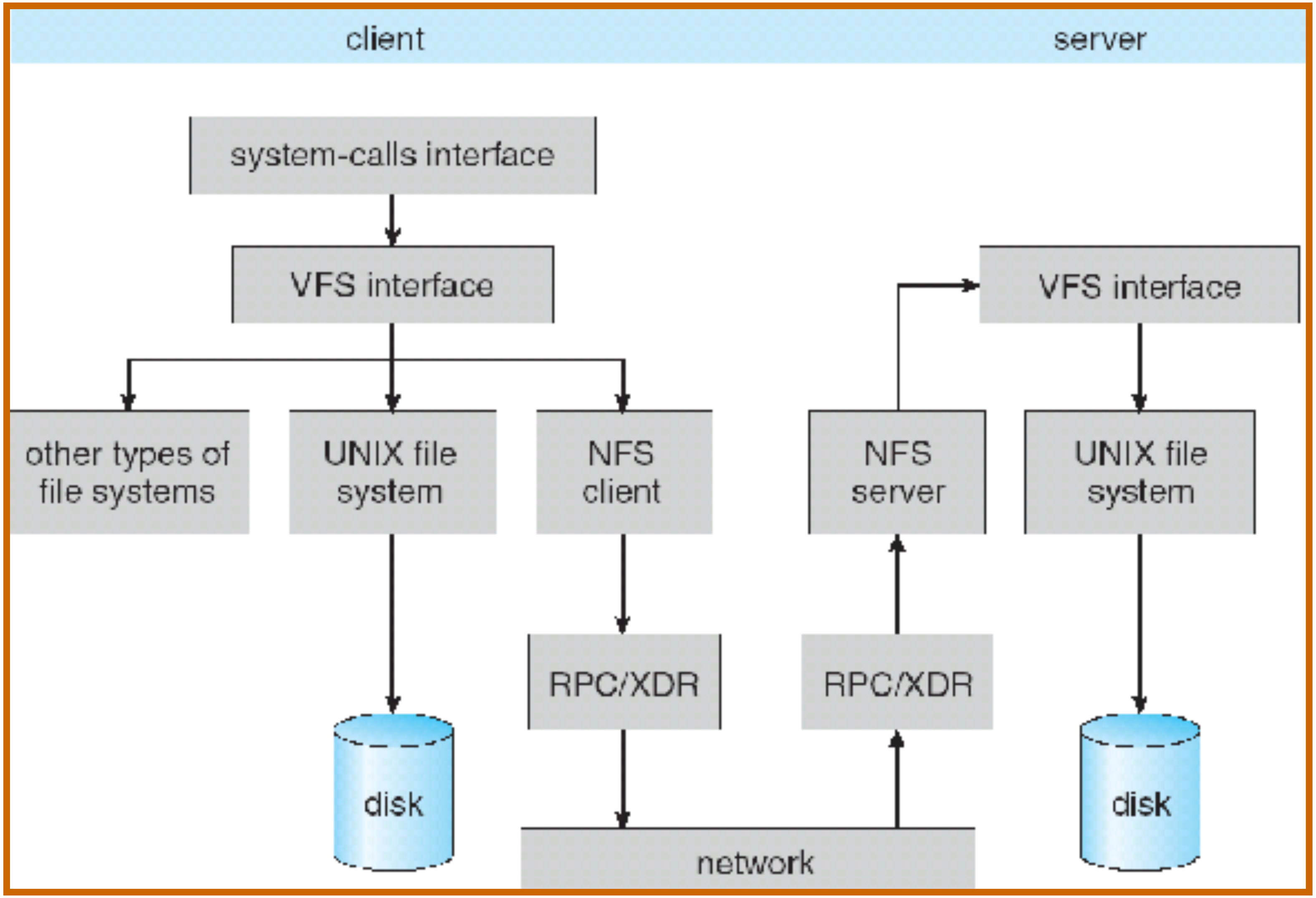
# Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

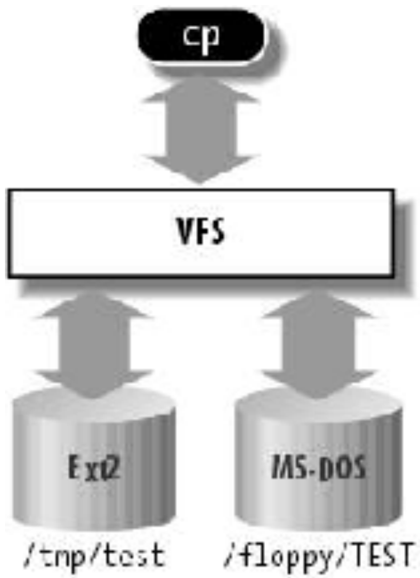
# Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: *go ask everyone "who has which files cached?"*
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

# Implementation of NFS



# Enabling Factor: Virtual Filesystem (VFS)




```
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);
```

- **VFS:** Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system
- In linux, "VFS" stands for "Virtual Filesystem Switch"

# Key Value Storage

- Handle huge volumes of data, e.g., PBs
  - Store (key, value) tuples
- Simple interface
  - put(key, value); // insert/write "value" associated with "key"
  - value = get(key); // get/read data associated with "key"
- Used sometimes as a simpler but more scalable "database"

# Key Values: Examples

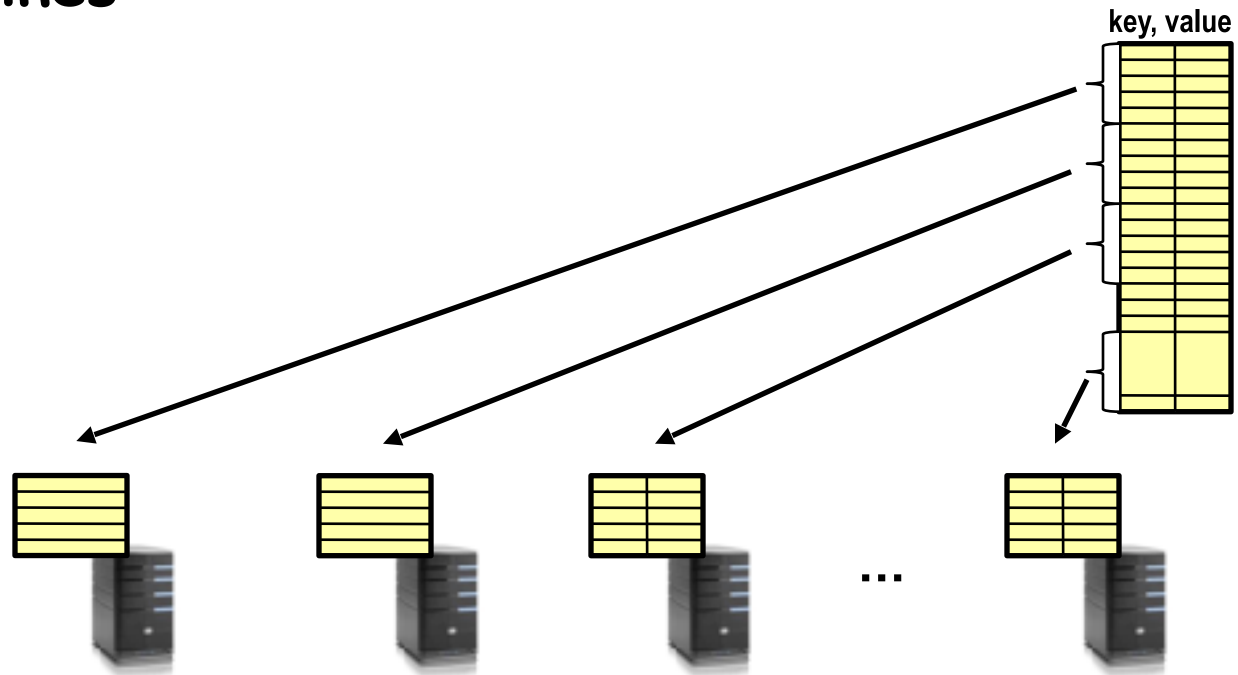
- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)
- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

# Key-value storage systems in real life

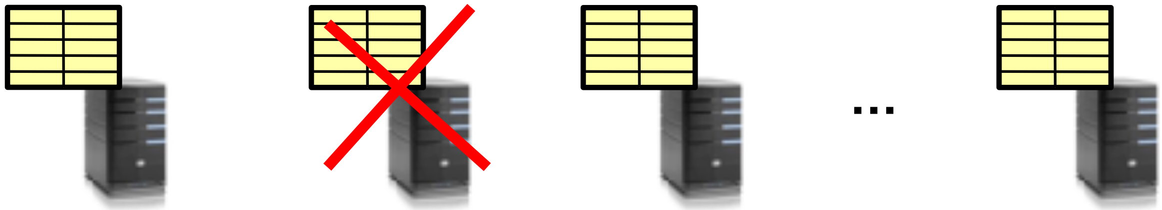
- Amazon
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- BigTable/HBase/Hypertable: distributed, scalable data storage
- Cassandra: "distributed data management system" (developed by Facebook)
- Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)
- eDonkey/eMule: peer-to-peer sharing system
- …

# Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines



# Challenges



- Fault Tolerance: handle machine failures without losing data  and without degradation in performance
- Scalability:
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- Consistency: maintain data consistency in face of node failures and message losses
- Heterogeneity (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s

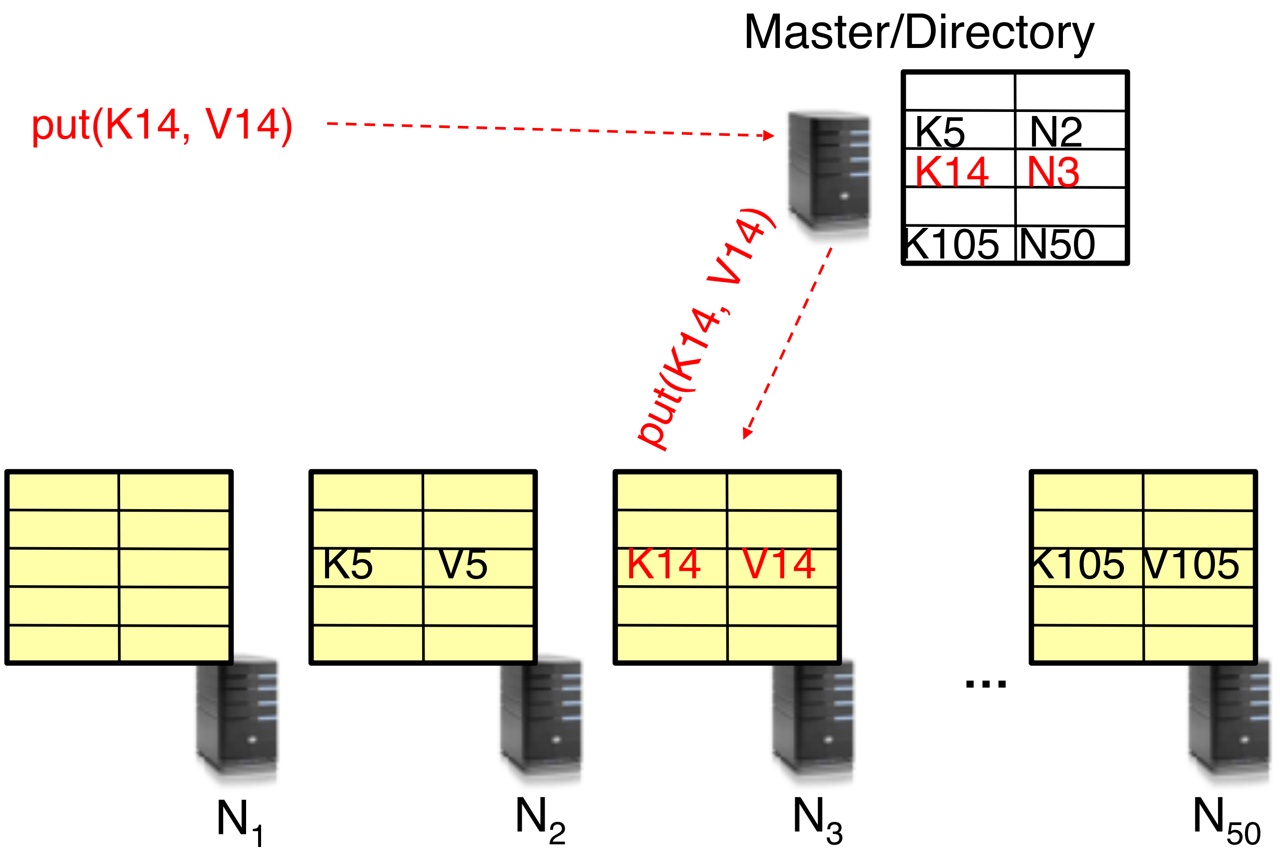
# Key Questions

- 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
  - Scalability
  - Consistency

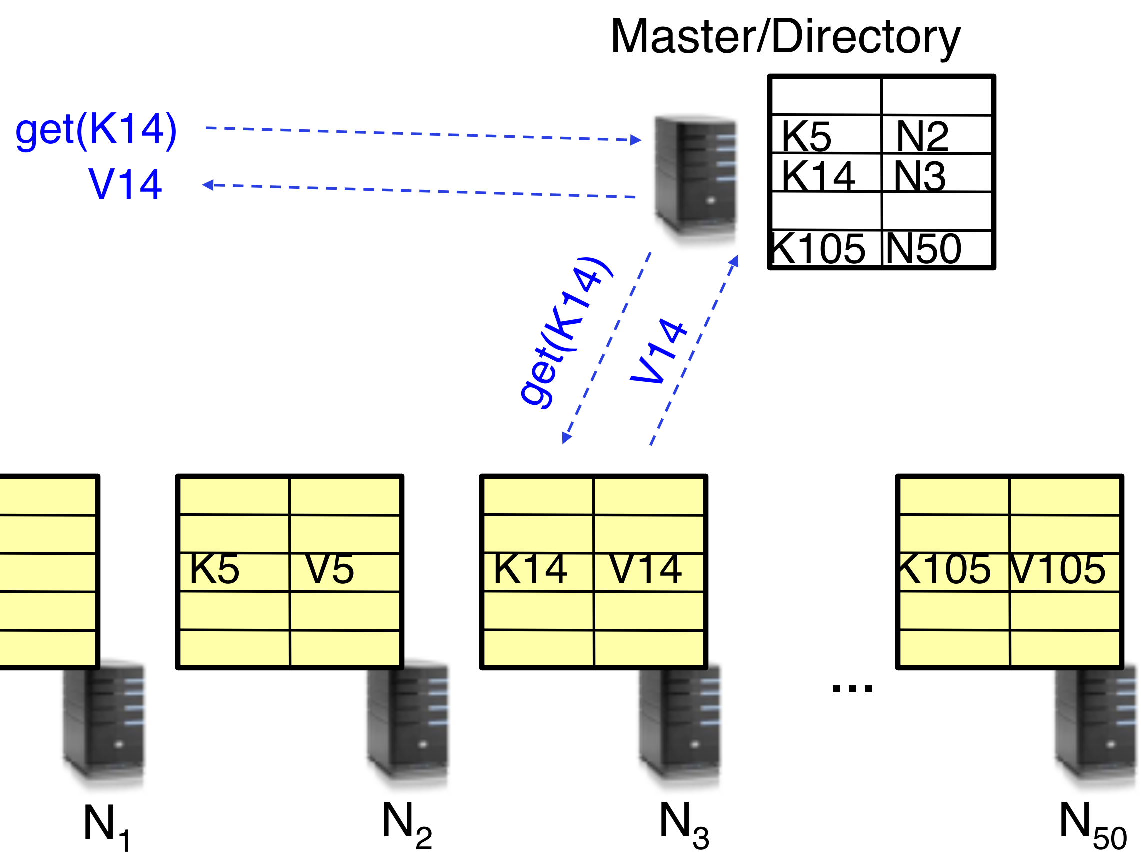
# Directory-Based Architecture

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



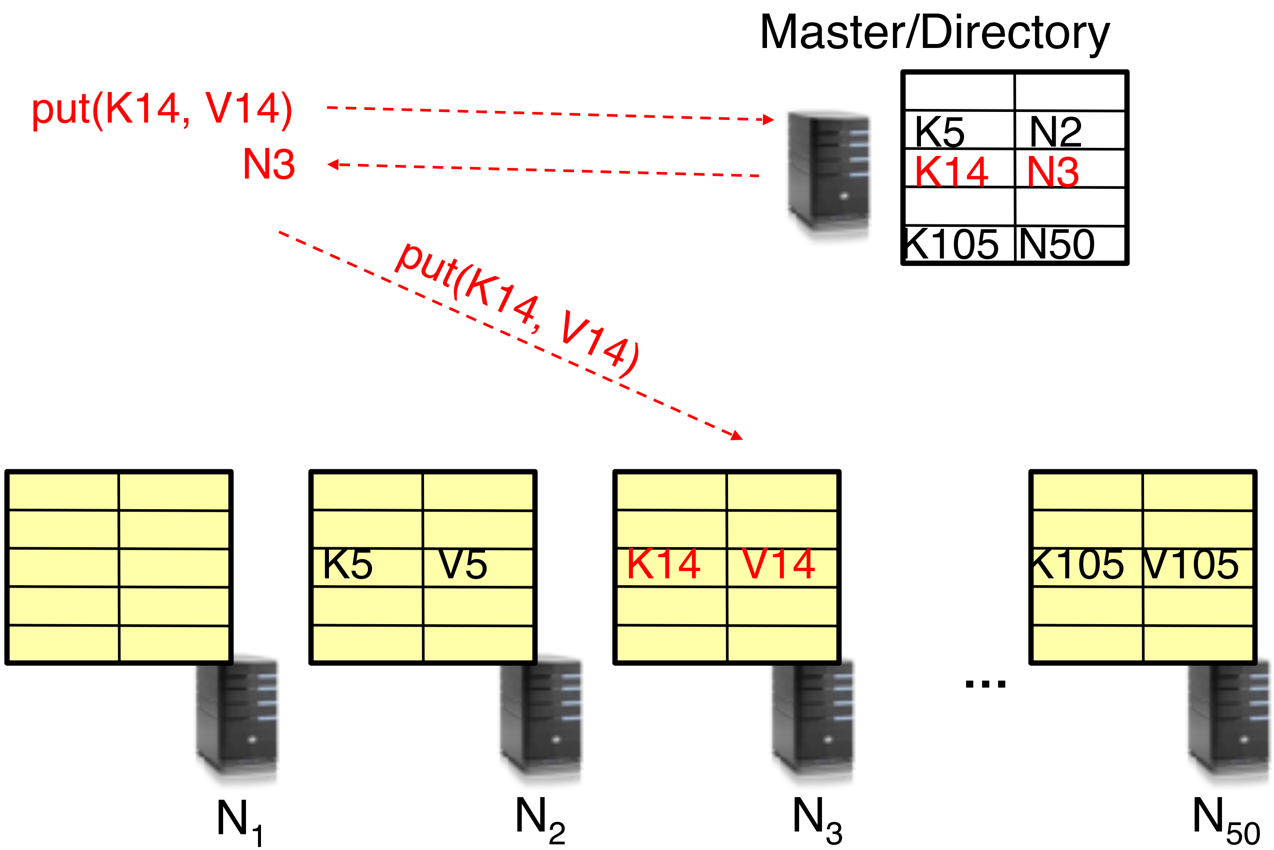
# Directory-Based Architecture

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



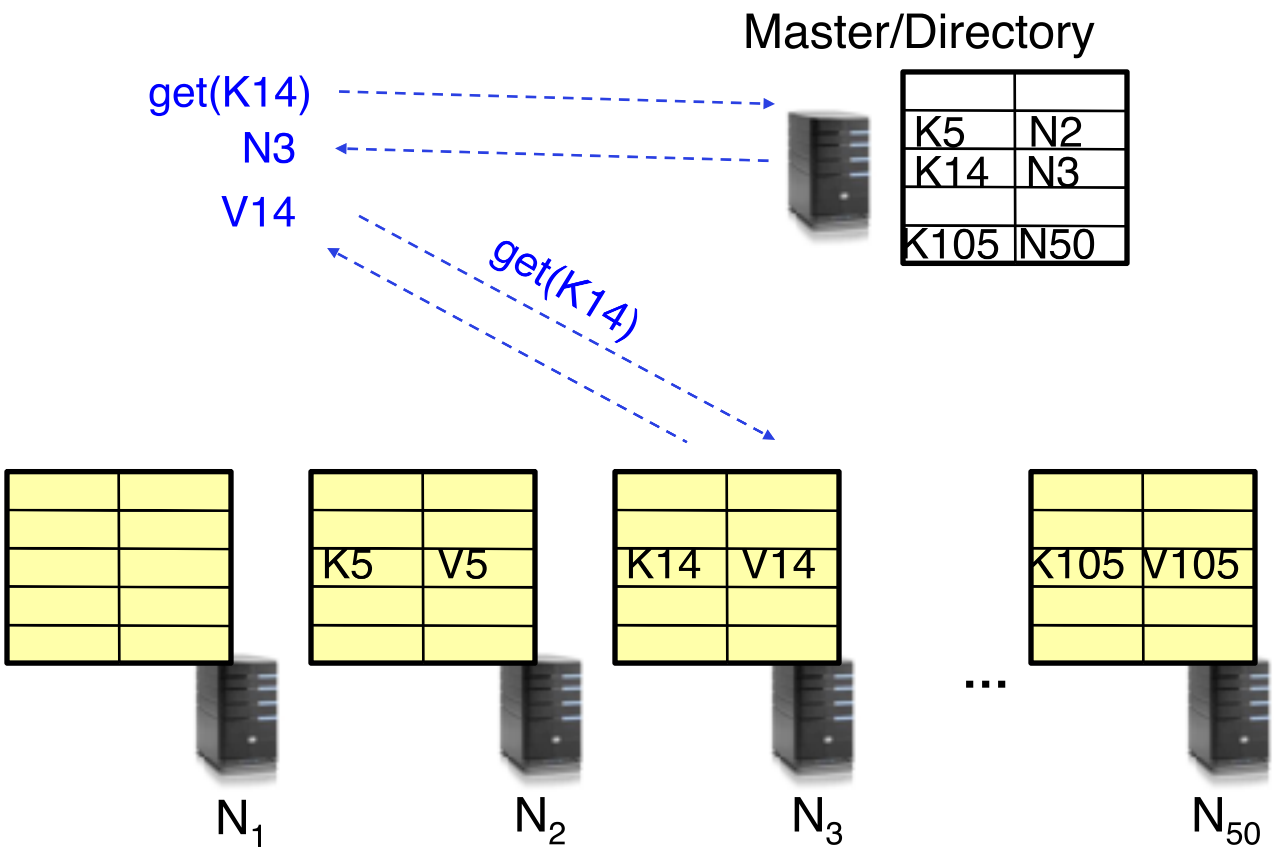
# Directory-Based Architecture

- Having the master relay the requests → recursive query
- Another method: iterative query (this slide)
  - Return node to requester and let requester contact node

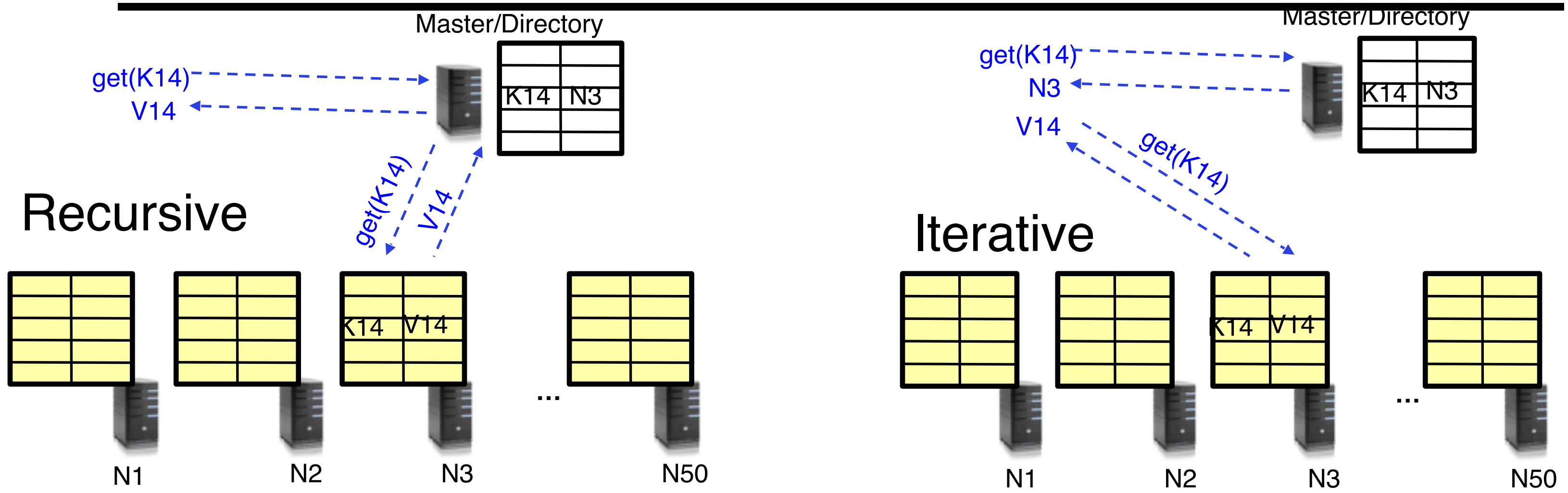


# Directory-Based Architecture

- Having the master relay the requests → recursive query
- Another method: iterative query
  - Return node to requester and let requester contact node



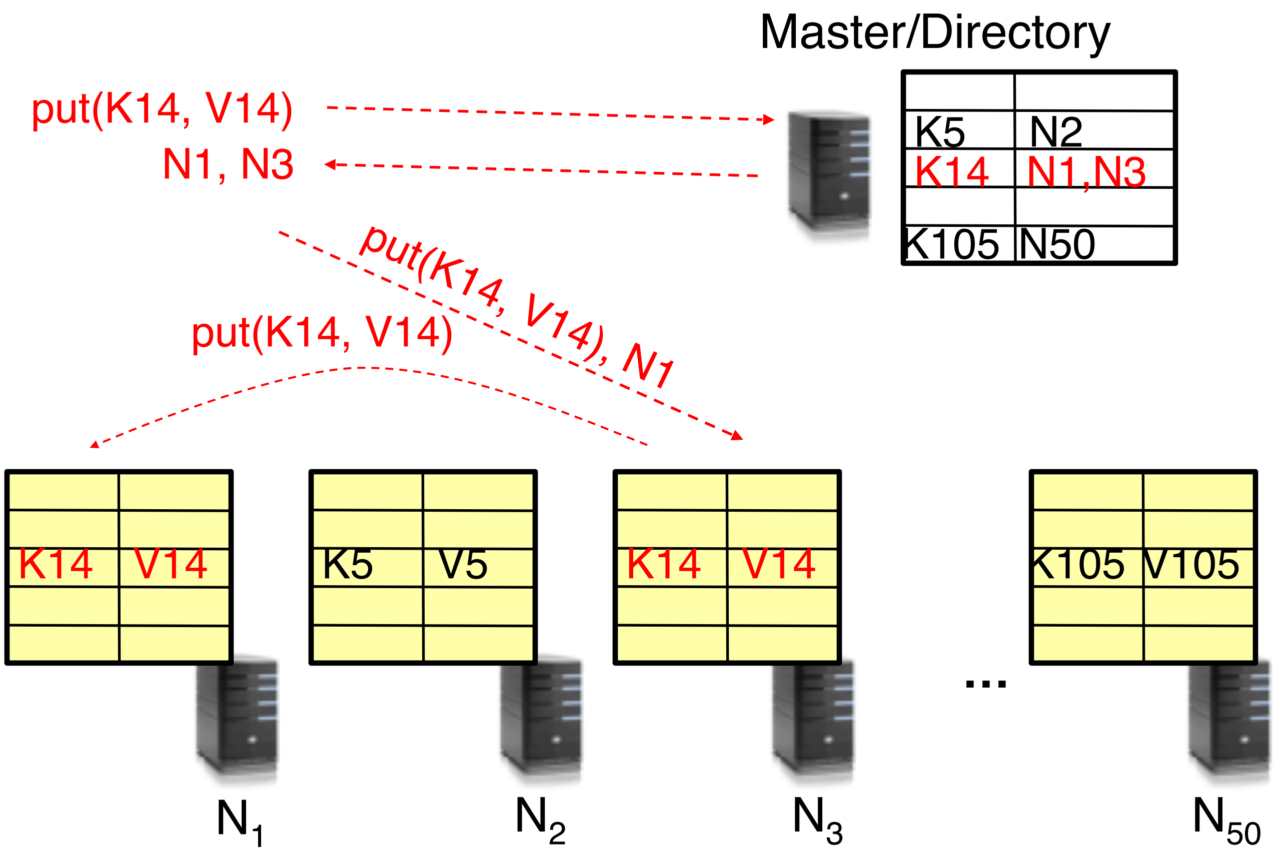
# Discussion: Iterative vs. Recursive Query




- **Recursive Query:**
  - Advantages:
    - » Faster, as typically master/directory closer to nodes
    - » Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Disadvantages: scalability bottleneck, as all "Values" go through master/directory
- **Iterative Query**
  - Advantages: more scalable
  - Disadvantages: slower, harder to enforce data consistency

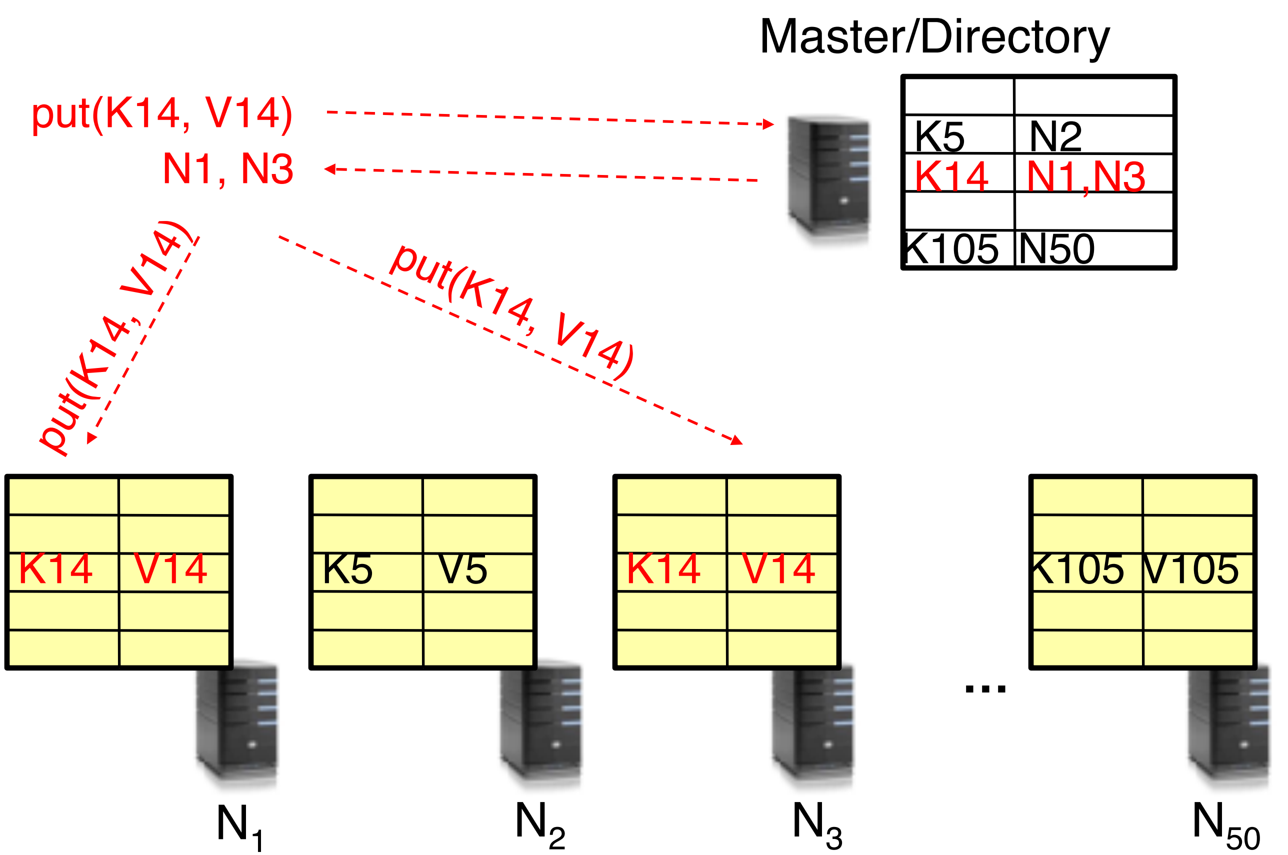
# Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



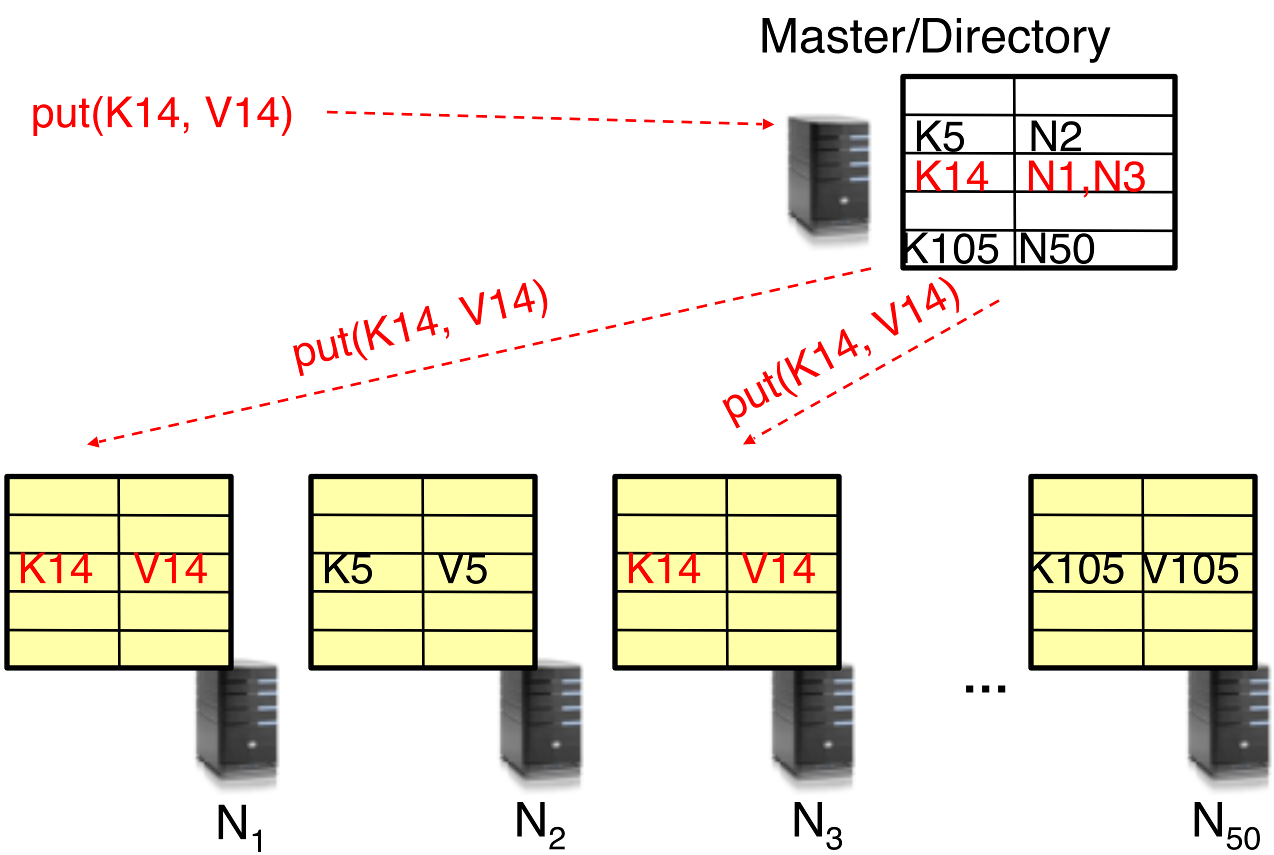
# Fault Tolerance

- Again, we can have
  - Recursive replication (previous slide)
  - Iterative replication (this slide)



# Fault Tolerance

- Or we can use recursive query and iterative replication…



# Scalability

- Storage: use more nodes
- Number of requests:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular value on more nodes
- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories
    - » How do you partition?

# Scalability: Load Balancing

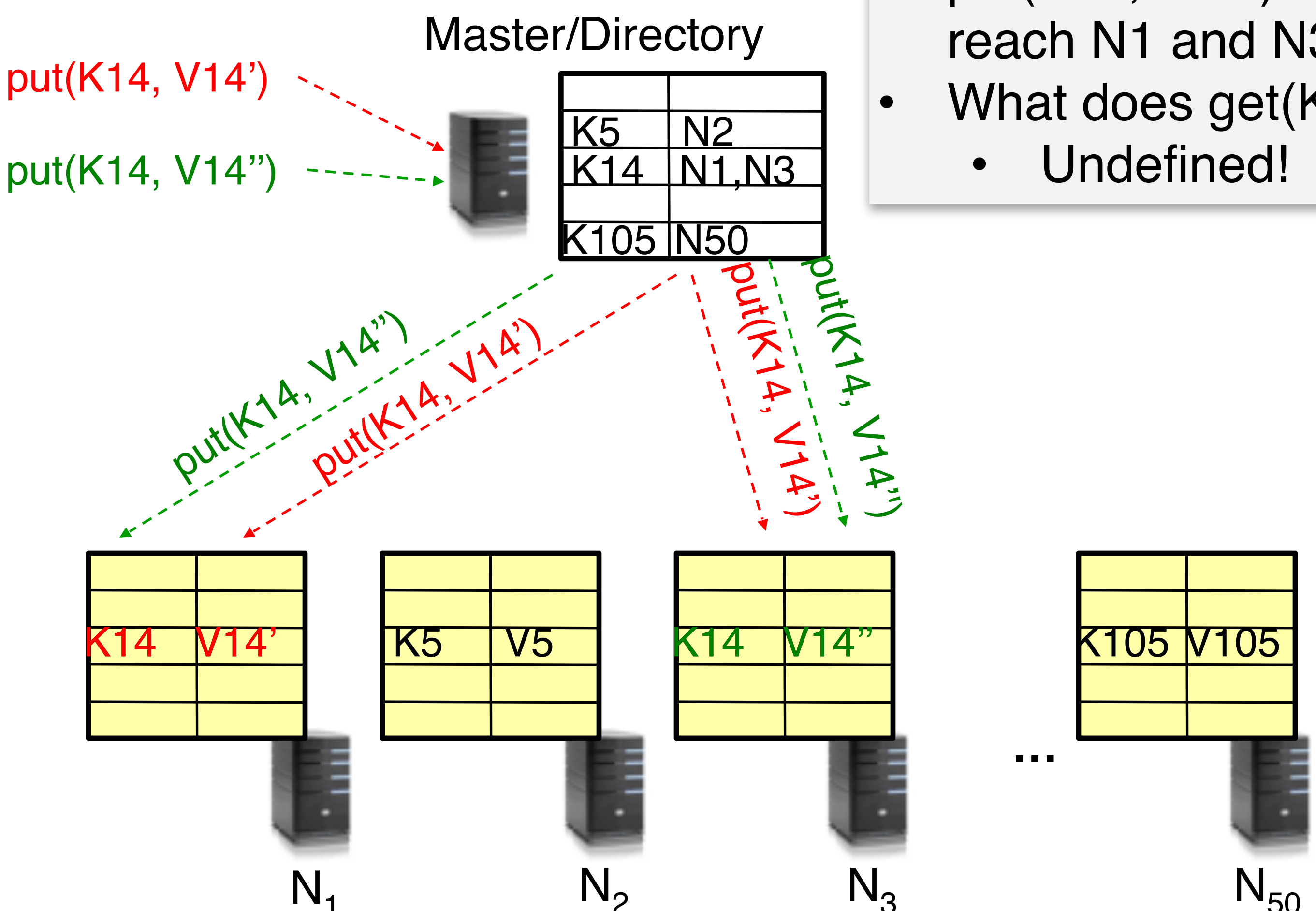
- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node
- What happens when a node fails?
  - Need to replicate values from fail node to other nodes

# Consistency

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

# Consistency (cont'd)

- **If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order**




- put(K14, V14') and put(K14, V14'') reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

# Consistency (cont'd)

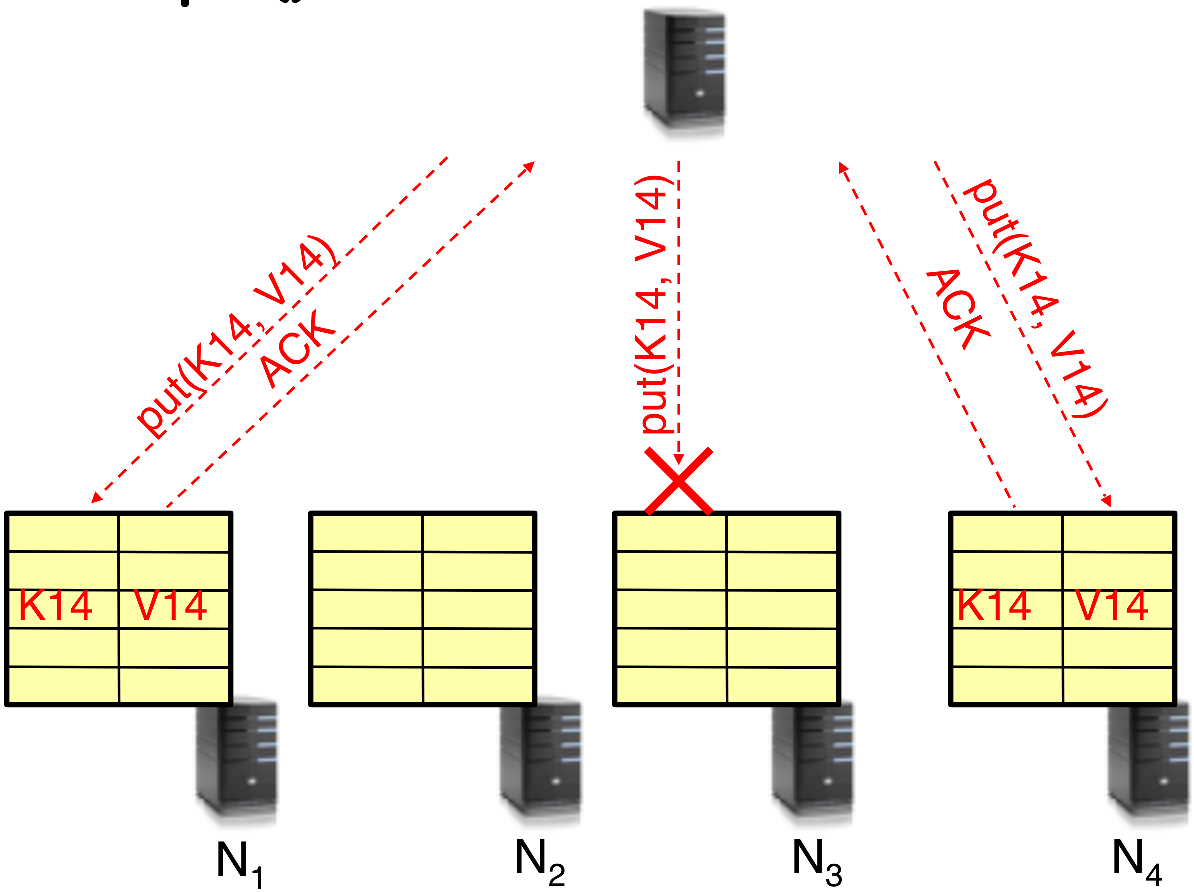
- Large variety of consistency models:
  - Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    - » Think "one updated at a time"
    - » Transactions
  - Eventual consistency: given enough time all updates will propagate through the system
    - » One of the weakest form of consistency; used by many systems in practice
  - And many others: causal consistency, sequential consistency, strong consistency, …

# Quorum Consensus

- Improve put() and get() operation performance
- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
  - get() waits for responses from at least R replicas
  - W+R > N
- Why does it work?
  - There is at least one node that contains the update
- Why might you use W+R > N+1?

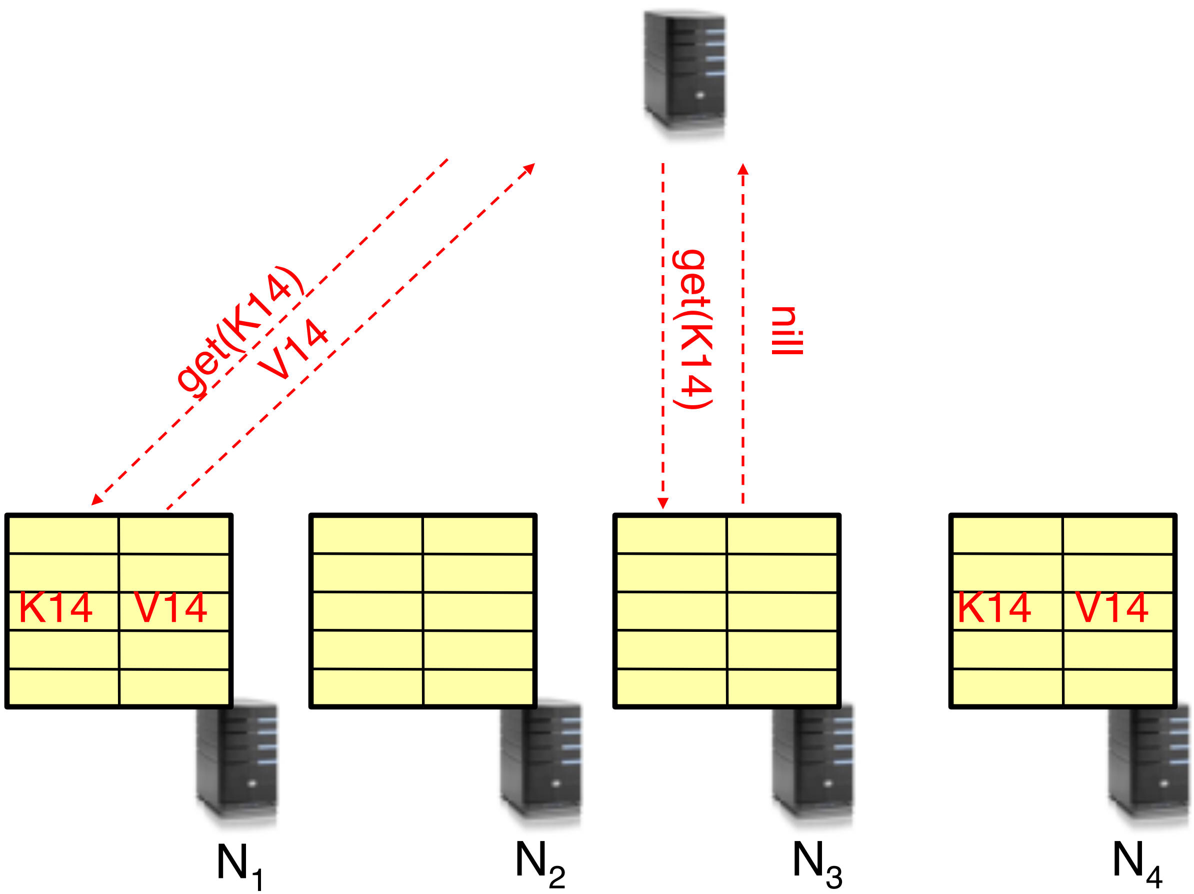
# Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}
- Assume put() on N3 fails



# Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer

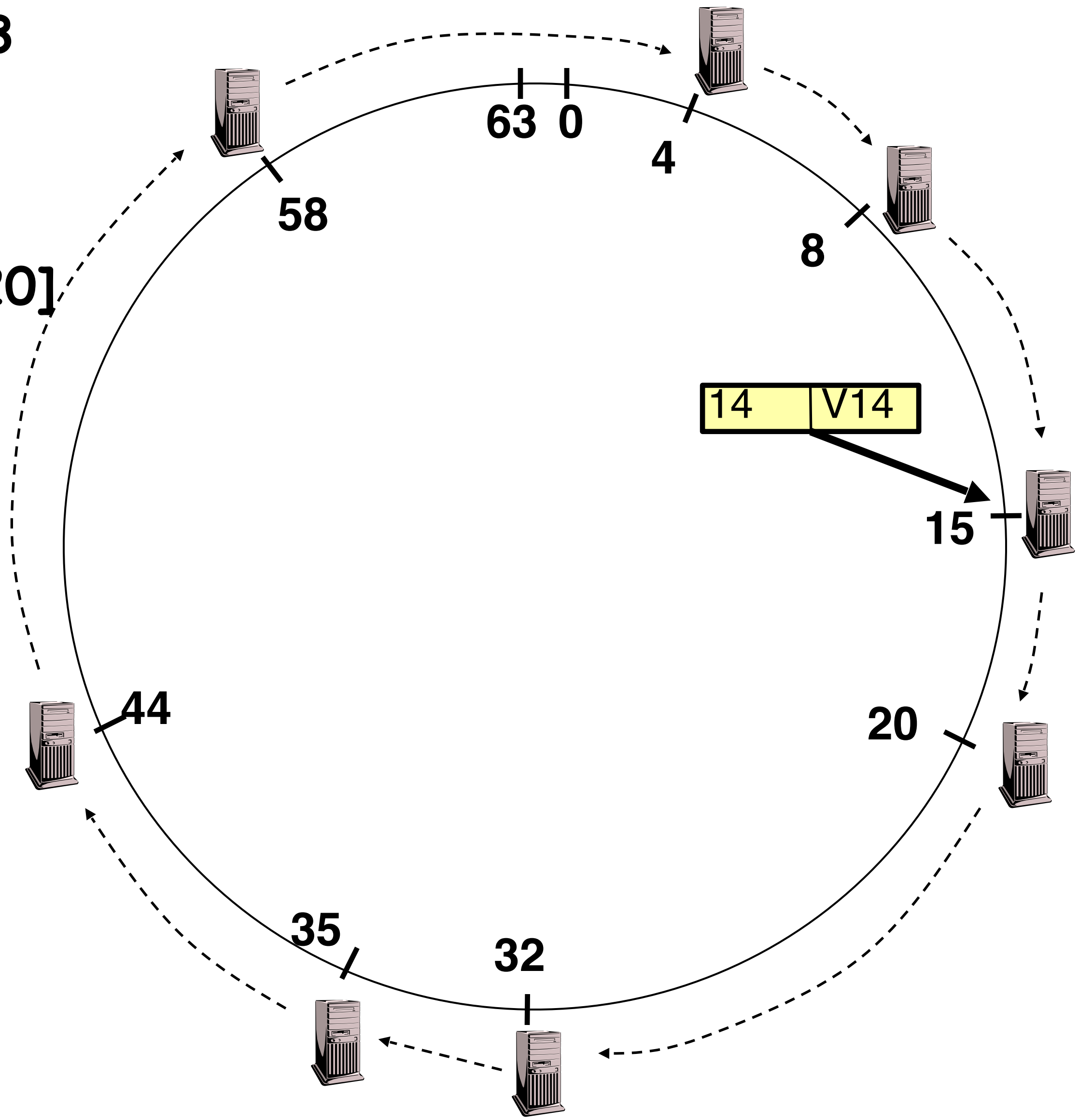


# Scaling Up Directory

- Challenge:
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!
- Solution: consistent hashing
- Associate to each node a unique id in an uni-dimensional space $0..2^m-1$
  - Partition this space across m machines
  - Assume keys are in same uni-dimensional space
  - Each (Key, Value) is stored at the node with the smallest ID larger than Key

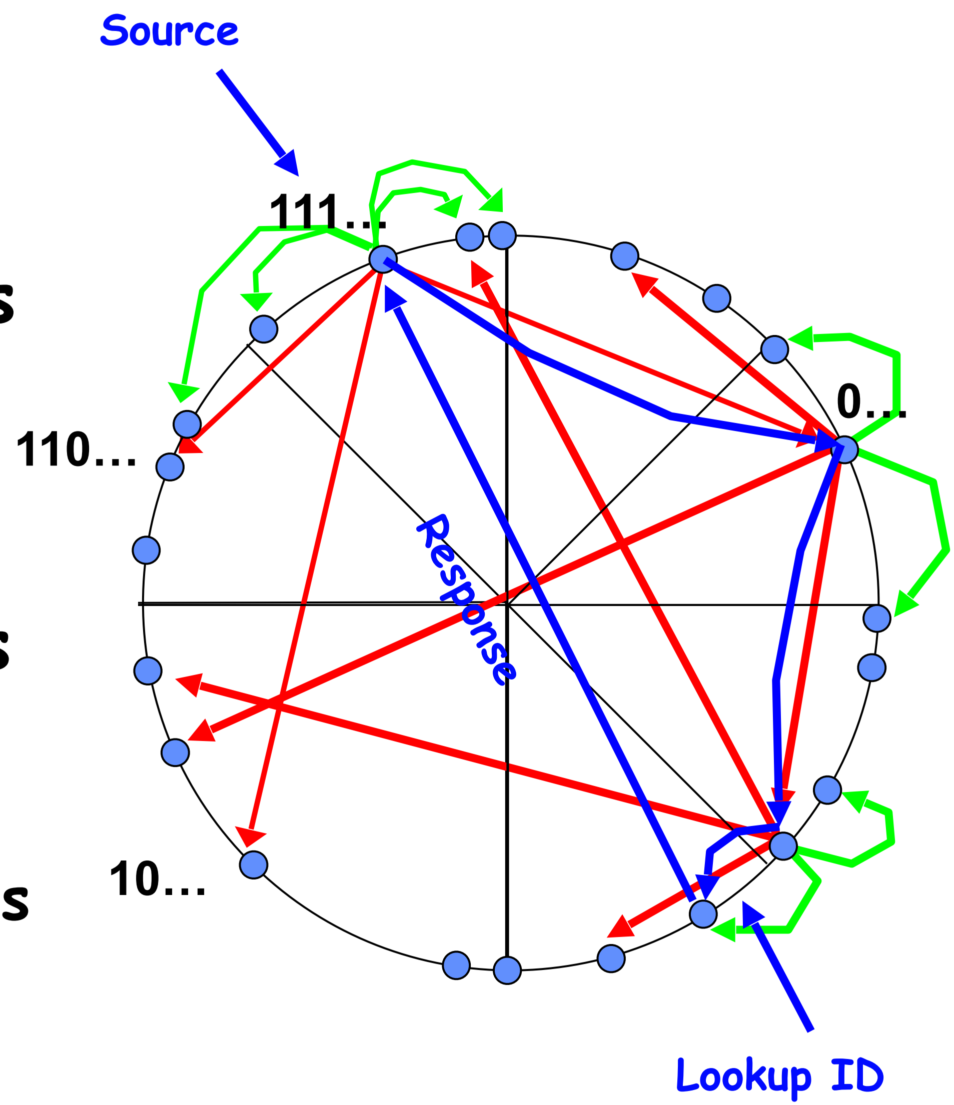
# Key to Node Mapping Example

- m = 6 → ID space: 0..63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16, 20]
- …
- Node 4 maps keys [59, 4]



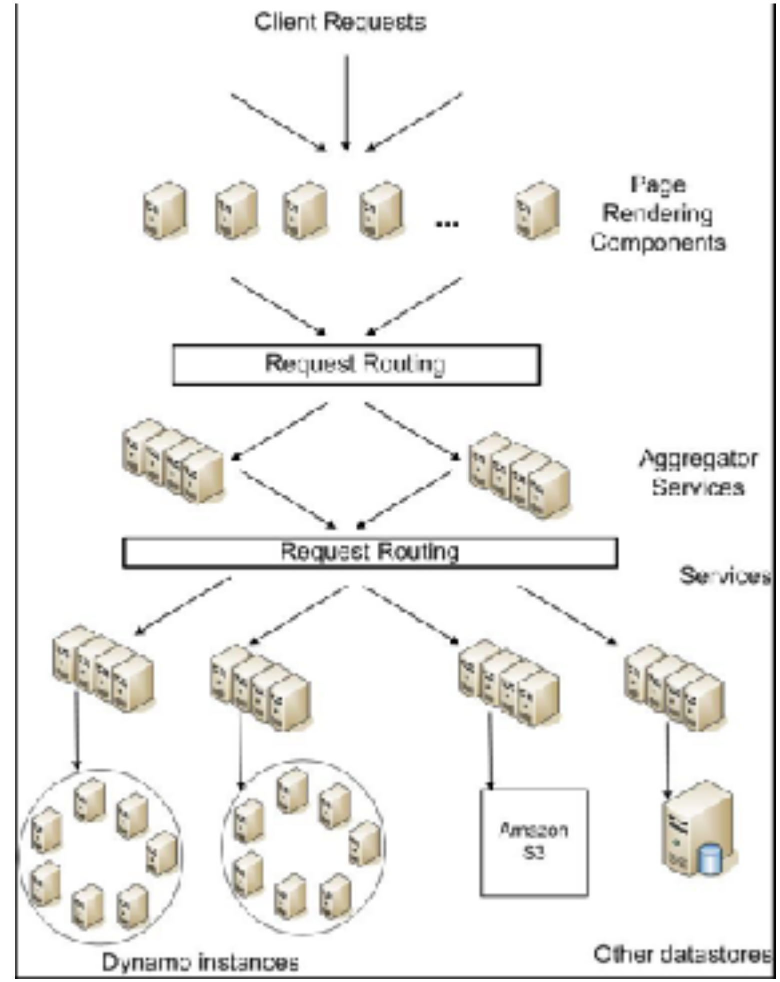
# Lookup in Chord-like system (with Leaf Set)

- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups
- Data Replication:
  - On leaf set



# DynamoDB Example: Service Level Agreements (SLA)

- Application can deliver its functionality in a bounded time:
  - Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second
- Contrast to services which focus on mean response time




Service-oriented architecture of Amazon's platform

# Summary (1/2)

- **Distributed File System:**
  - Transparent access to files stored on a remote disk
  - Caching for performance
- **Cache Consistency:** Keeping client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks to be notified by server of changes
- **Remote Procedure Call (RPC):** Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments (in stub)
- **VFS:** Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems

# Summary (2/2)

- **Key-Value Store:**
  - Two operations
    - » put(key, value)
    - » value = get(key)
  - Challenges
    - » Fault Tolerance → replication
    - » Scalability → serve get()'s in parallel; replicate/cache hot tuples
    - » Consistency → quorum consensus to improve put() performance