CS162
Operating Systems and Systems Programming
Lecture 25

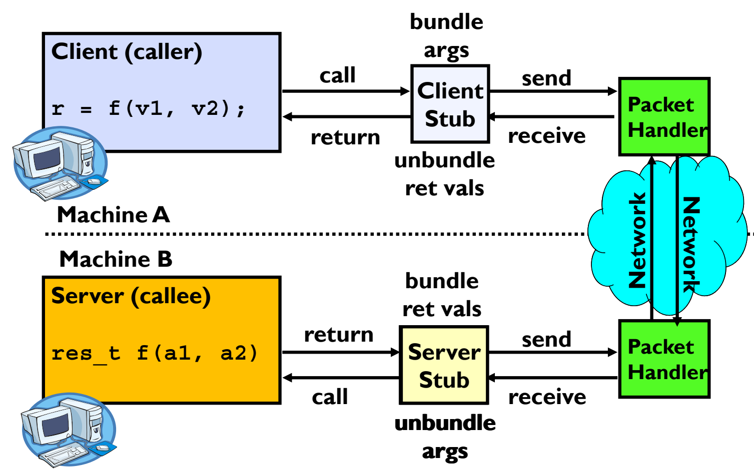
Distributed Storage, NFS and AFS, Key Value Stores

November 29th, 2020

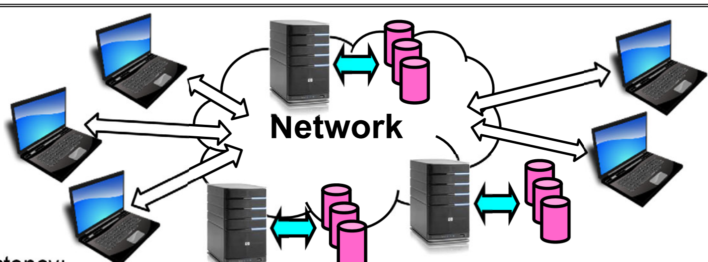
Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

Recall: RPC Information Flow

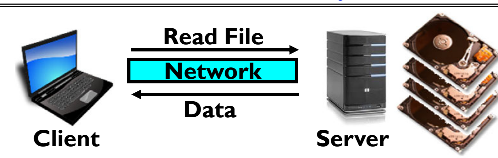


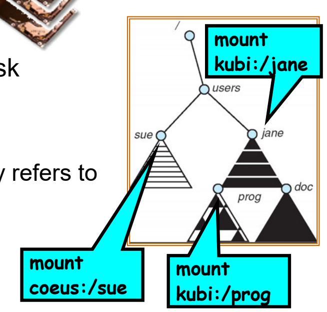
Recall: Network-Attached Storage and the CAP Theorem



- 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”

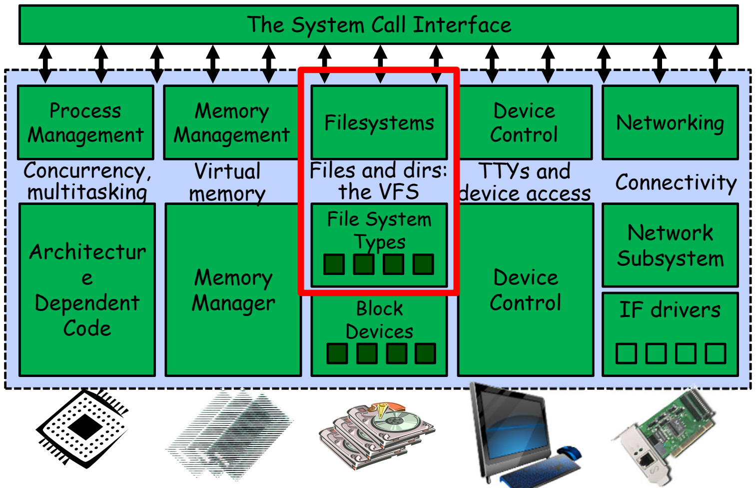
Distributed File Systems



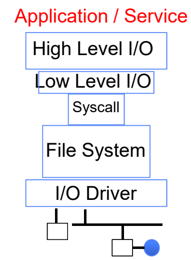


- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `kubi.cs.berkeley.edu`
- *Naming* Choices:
  - [Hostname,localname]: Filename includes server
    - » No location or migration transparency, except through DNS remapping
  - A global name space: Filename unique in “world”
    - » Can be served by any server

Enabling Design: VFS



Recall: Layers of I/O…



User App:

```
length = read(input_fd, buffer, BUFFER_SIZE);
```

User library:

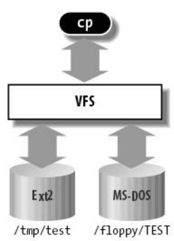
```
ssize_t read(int, void *, size_t) {
  marshal args into registers
  issue syscall
  register result of syscall to rtn value
};
```

```
Exception U→K, interrupt processing
void syscall_handler (struct intr_frame *f) {
   unmarshall call#, args from regs
   dispatch : handlers[call#](args)
   marshal results fo syscall ret
}
```

```
ssize_t vfs_read(struct file *file, char __user *buf,
                 size_t count, loff_t *pos) {
   User Process/File System relationship
   call device driver to do the work
}
```

Device Driver

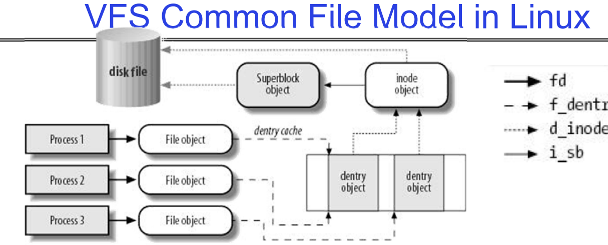
Virtual Filesystem Switch




```
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);
```

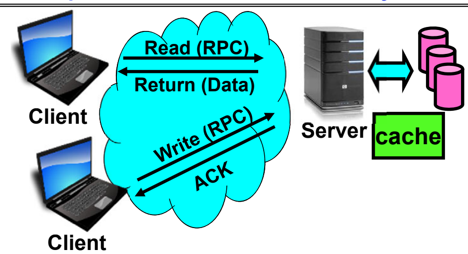
- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used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

VFS Common File Model in Linux



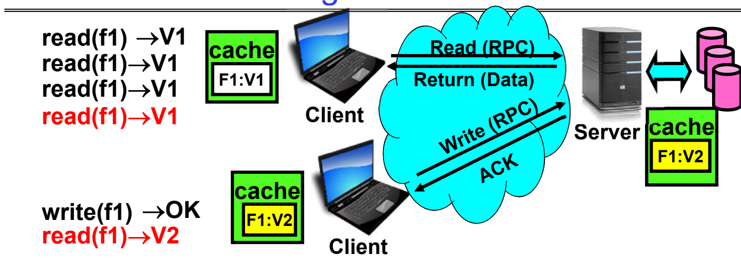
- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS treats directories as files)
- May need to fit the model by faking it
  - Example: make it look like directories are files
  - Example: make it look like have inodes, superblocks, etc.

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, but can be cache at server-side
- Advantage: Server provides 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

Dealing with Failures

- What if server crashes? Can client wait until it comes back and just continue making requests?
  - Changes in server's cache but not in disk are lost
- What if there is shared state across RPC's?
  - Client opens file, then does a seek
  - Server crashes
  - What if client wants to do another read?
- Similar problem: What if client removes a file but server crashes before acknowledgement?

Stateless Protocol

- 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)
- Recall HTTP: Also a stateless protocol
  - Include cookies with request to simulate a session

Administrivia

- Midterm 3: Thursday: 5-7PM as before
  – Material up to Lecture 25 (Today's lecture)
  – Cameras and Zoom screen sharing again as with Midterm 2
  – No excuse to not have camera and screen sharing turned on!
- Review session Tuesday (tomorrow): 7-9pm
  – Zoom link should be published on Piazza
- Lecture 26 will be a fun lecture
  – Let me know if there are topics you would like to discuss!
  – Not responsible for contents of Wednesday's lecture on Midterm 3!

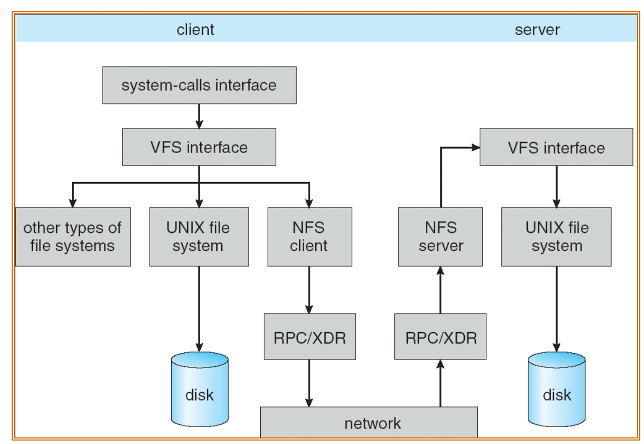
Case Study: Network File System (NFS)

- 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
  – XDR Serialization standard for data format independence
  – 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

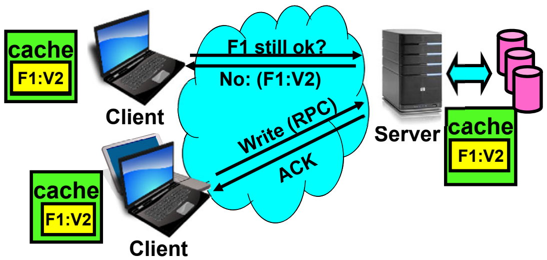
- NFS servers are stateless; each request provides all arguments require for execution
  – E.g. reads include information for entire operation, such as **`ReadAt(inumber,position)`**, not **`Read(openfile)`**
  – No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing them 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 other 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 Architecture



NFS Cache consistency

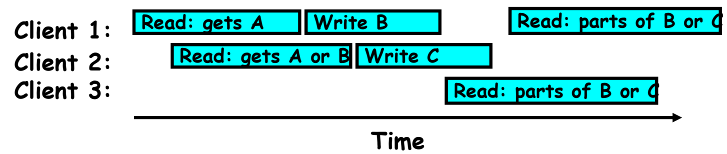
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version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

What about: Sharing Data, rather than Files ?

- Key:Value stores are used everywhere
- Native in many programming languages
  - Associative Arrays in Perl
  - Dictionaries in Python
  - Maps in Go
  - …
- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

Key Value Storage

Simple interface

- **put(key, value);** `// Insert/write "value" associated with key`
- **get(key);** `// Retrieve/read value associated with key`

Why Key Value Storage?

- Easy to Scale
  - Handle huge volumes of data (e.g., petabytes)
  - Uniform items: distribute easily and roughly equally across many machines
- Simple consistency properties
- Used as a simpler but more scalable "database"
  - Or as a building block for a more capable DB

Key Values: Examples

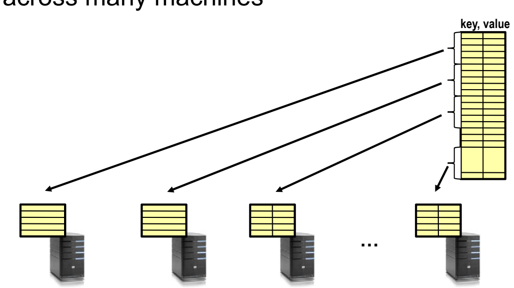
- Amazon:
  - Key: customerID
  - Value: customer profile (e.g ;redit card, ..)
- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

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: simplify storage interface (i.e. put/get), then partition set of key-values across many machines



Challenges

- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **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

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?

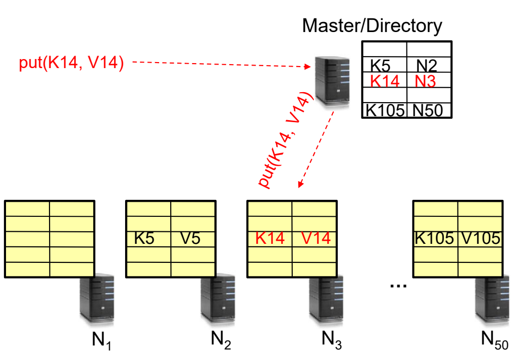
- And, do the above while providing
  - Scalability
  - Fault Tolerance
  - Consistency

How to solve the "where?"

- Hashing to map key space ⇒ location
  - But what if you don't know all the nodes that are participating?
  - Perhaps they come and go …
  - What if some keys are really popular?
- Lookup
  - Hmm, won't this be a bottleneck and single point of failure?

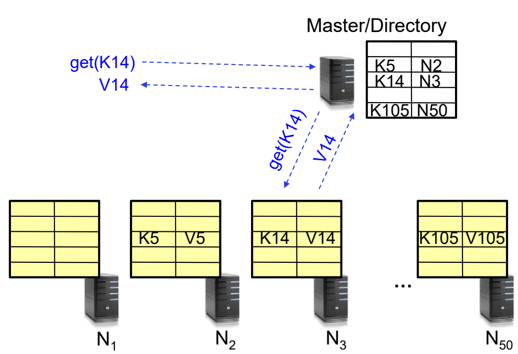
Recursive Directory Architecture (put)

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



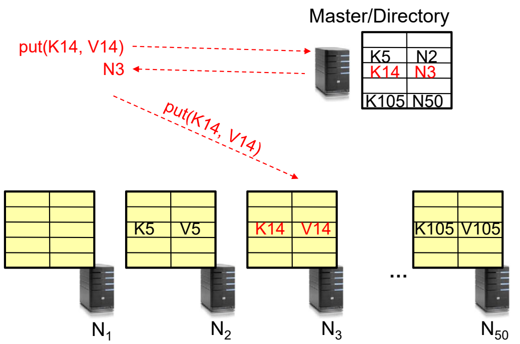
Recursive Directory Architecture (get)

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



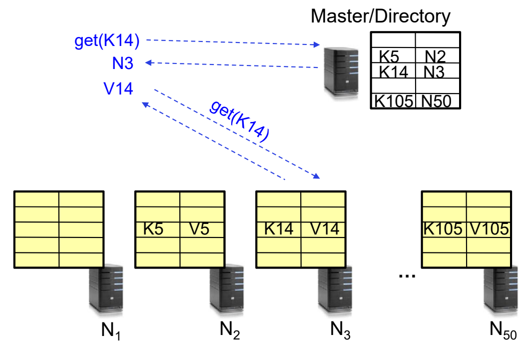
Iterative Directory Architecture (put)

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

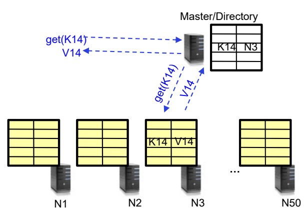


Iterative Directory Architecture (get)

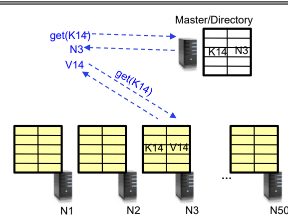
- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  – Return node to requester and let requester contact node



Iterative vs. Recursive Query



Recursive



Iterative

+ 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

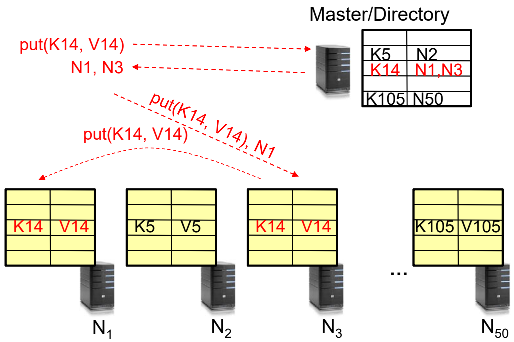
+ More scalable, clients do more work
- Harder to enforce consistency

Scalability: Is it easy to make the system bigger?

- 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 item on more nodes
- Master/Directory Scalability
  – Replicate It (multiple identical copies)
  – Partition it, so different keys are served by different directories
    » But how do we do this….?

Fault Tolerance

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

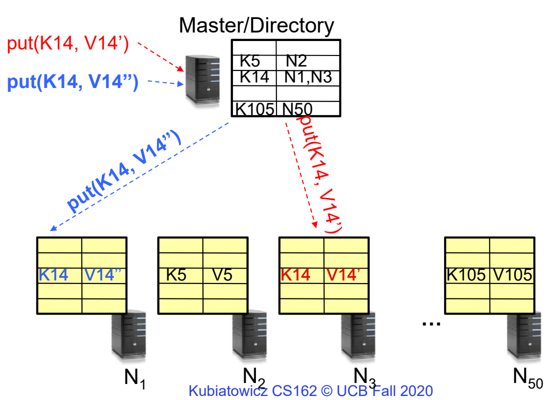


Consistency

- 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)

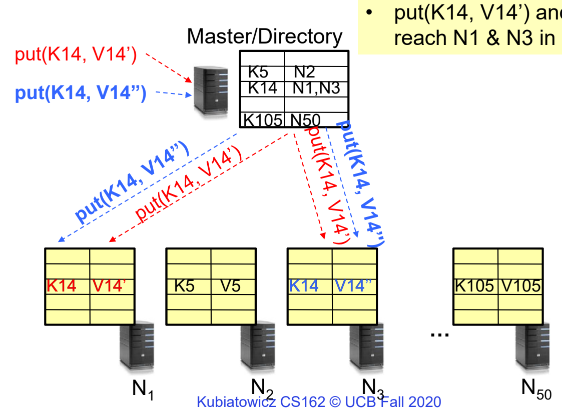
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



Consistency (cont'd)

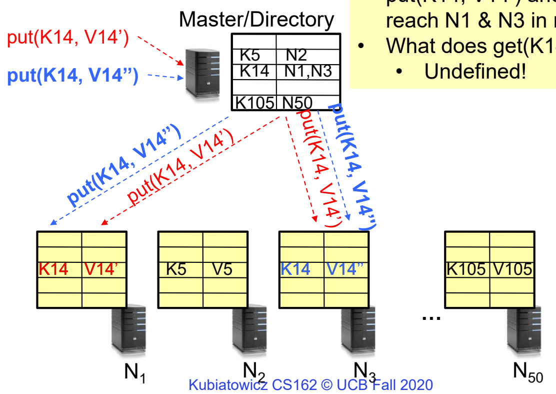
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order

- put(K14, V14') and put(K14, V14'') reach N1 & N3 in reverse order!



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 & N3 in reverse order!
- What does get(K14) return?
  - Undefined!

Large Variety of Consistency Models

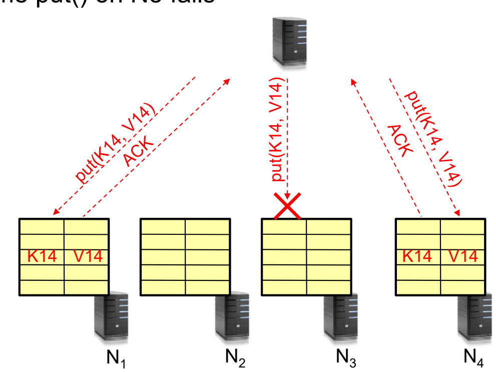
- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  - Think “one updated at a time”
  - Transactions
- Eventual consistency: given enough time all updates will propagate through the system
  - One of the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

Quorum Consensus

- Improve put() and get() operation performance
  - In the presence of replication!
- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
    - » Different updates need to be differentiated by something monotonically increasing like a timestamp
    - » Allows us to replace old values with updated ones
  - 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?

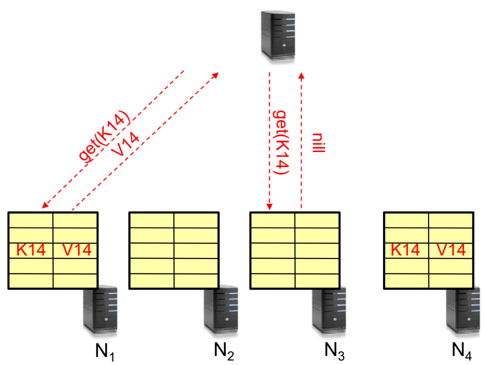
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}
- Assume put() on N3 fails



Quorum Consensus Example

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



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

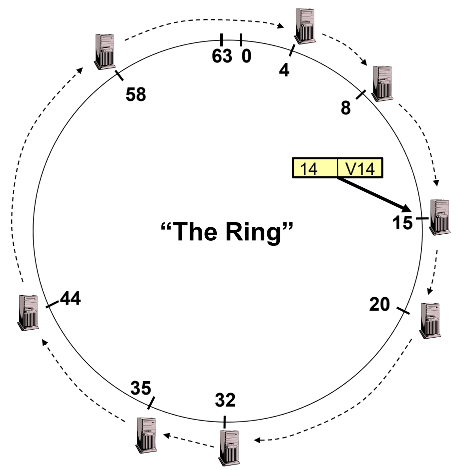
- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node
- What happens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

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**
  - **Provides mechanism to divide [key,value] pairs amongst a (potentially large!) set of machines (nodes) on network**
- Associate to each node a unique *id* in an *uni*-dimensional space $0..2^m-1$
  $\Rightarrow$ Wraps around: Call this "the ring!"
  - Partition this space across *n* 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

- Paritioning example with 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]
- For this example, the mapping [14, V14] maps to node with ID=15
  - Node with smallest ID larger than 14 (the key)
- In practice, m=256 or more!
  - Uses cryptographically secure hash such as SHA-256 to generate the node IDs

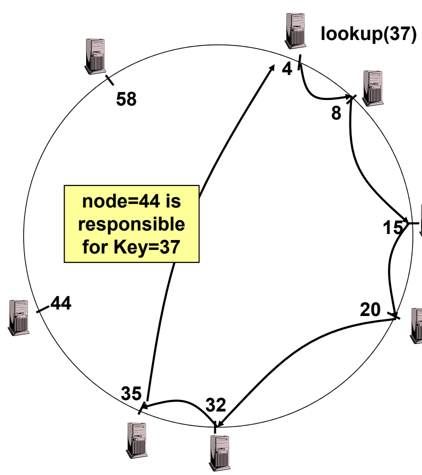


Chord: Distributed Lookup (Directory) Service

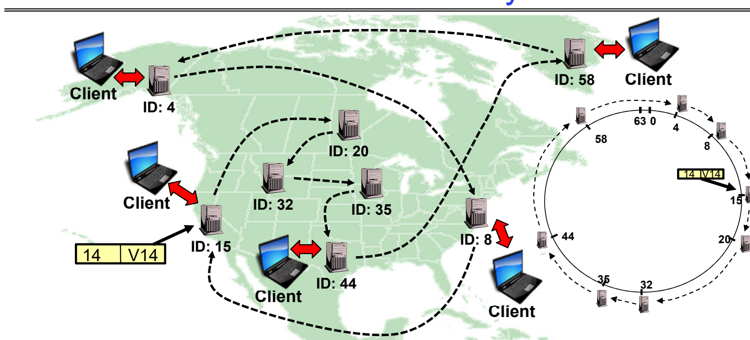
- "Chord" is a Distributed Lookup Service
  - Designed at MIT and here at Berkeley (Ion Stoica among others)
  - Simplest and cleanest algorithm for distributed storage
    - » Serves as comparison point for other optims
- Import aspect of the design space:
  - Decouple correctness from efficiency
  - 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!

Chord's Lookup Mechanism: Routing!

- 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 much better normal lookup time is O(log n)
  - Dynamic performance optimization (finger table mechanism)
    - » More later!!!



But what does this really mean??



- Node names intentionally scrambled WRT geography!
  - Node IDs generated by secure hashes over metadata
    - » Including things like the IP address
  - This geographic scrambling spreads load and avoids hotspots
- Clients access distributed storage through any member of the network

Stabilization Procedure

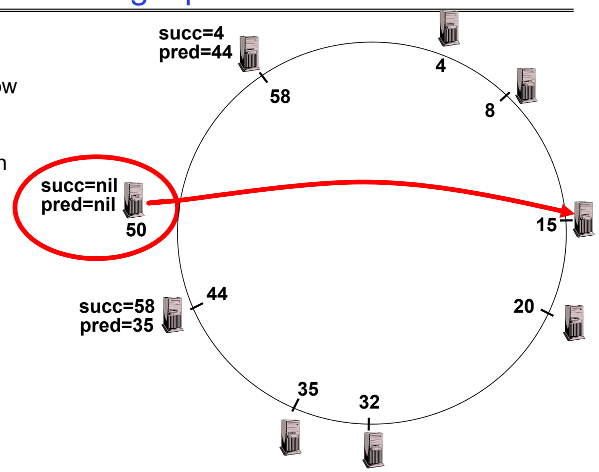
- Periodic operation performed by each node n to maintain its successor when new nodes join the system
  - The primary Correctness constraint

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
      succ = x;     // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
      pred = n';     // if n' is better predecessor, update
```

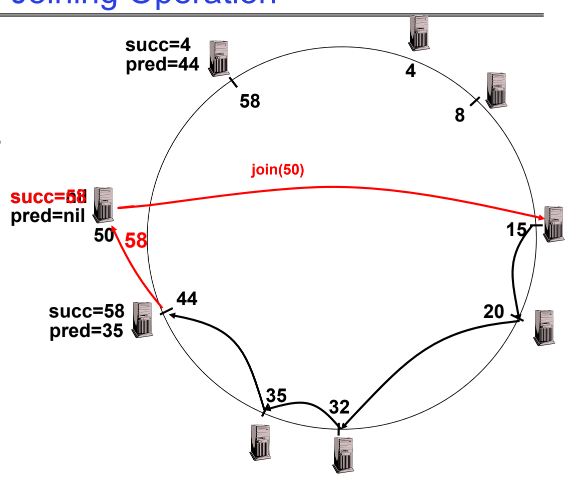

Joining Operation

- Node with id=50 joins the ring
- Node 50 must know at least one node already in system
  - Assume known node is 15



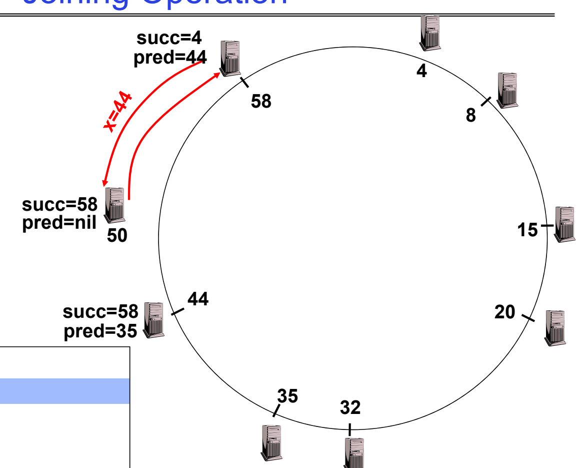
Joining Operation

- n=50 sends join(50) to node 15
  - Join propagated around ring!
- n=44 returns node 58
- n=50 updates its successor to 58



Joining Operation

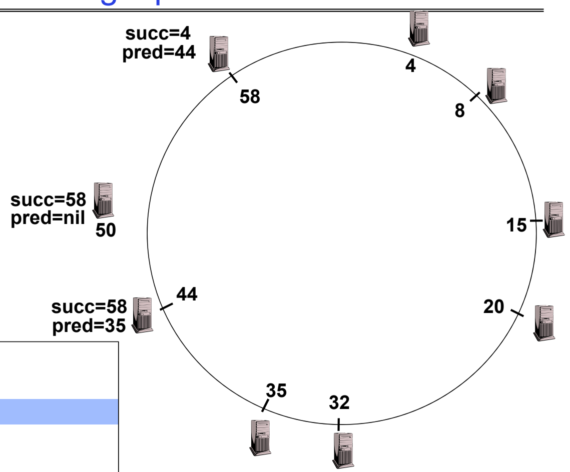
- n=50 executes stabilize()
- n's successor (58) returns x = 44




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

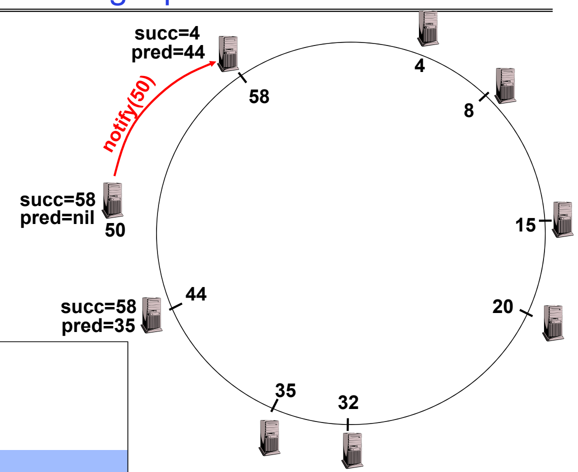
- n=50 executes stabilize()
  - x = 44
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

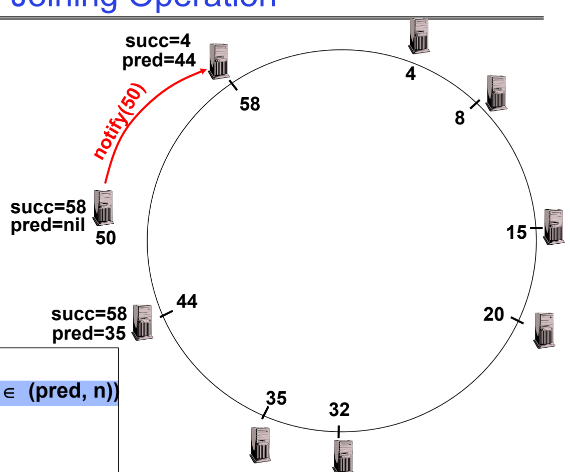
- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it's successor (58) notify(50)




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

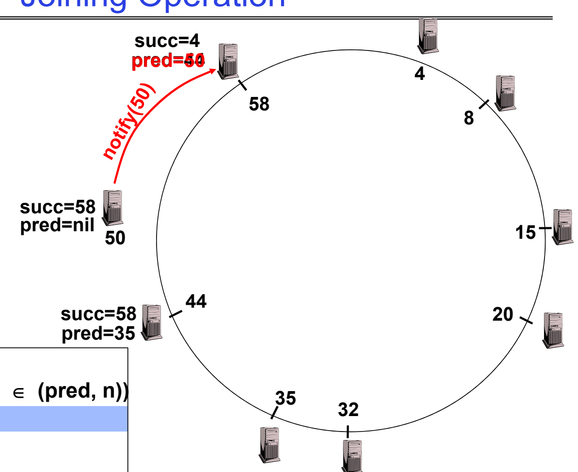
- n=58 executes notify(50)
  - pred = 44
  - n' = 50




```
n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n'
```


Joining Operation

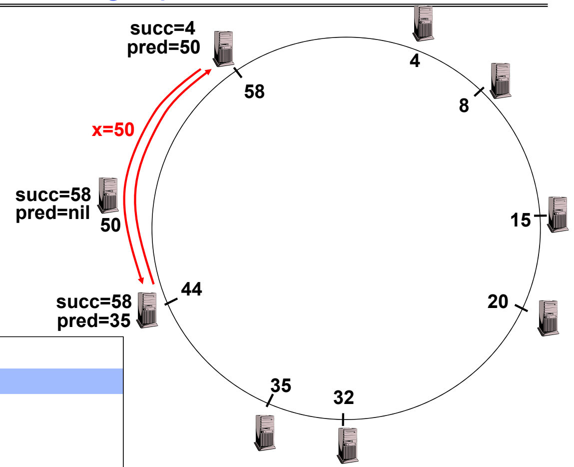
- n=58 executes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50




```
n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n'
```

Joining Operation

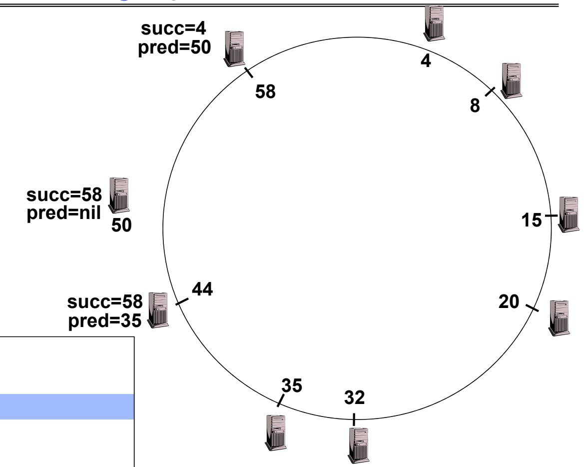
- n=44 executes stabilize()
- n's successor (58) returns x=50




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
     succ = x;
  succ.notify(n);
```


Joining Operation

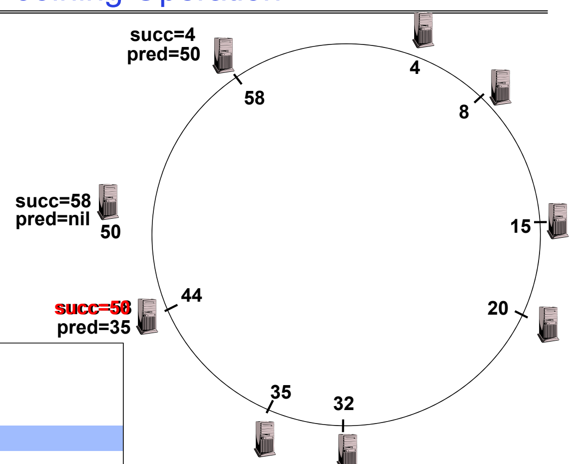
- n=44 executes stabilize()
  - x=50
  - succ=58




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
     succ = x;
  succ.notify(n);
```


Joining Operation

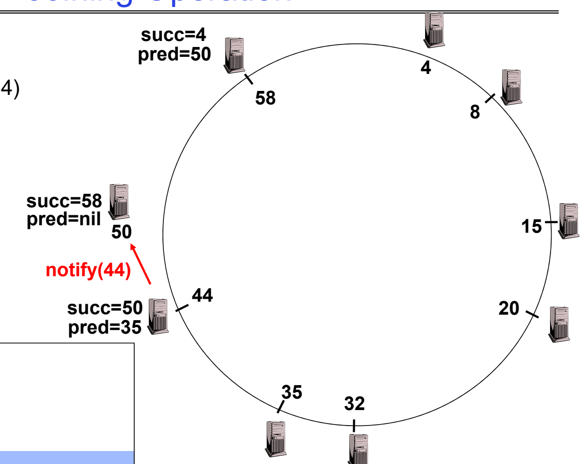
- n=44 executes stabilize()
  - x=50
  - succ=58
- n=44 sets succ=50




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
     succ = x;
  succ.notify(n);
```


Joining Operation

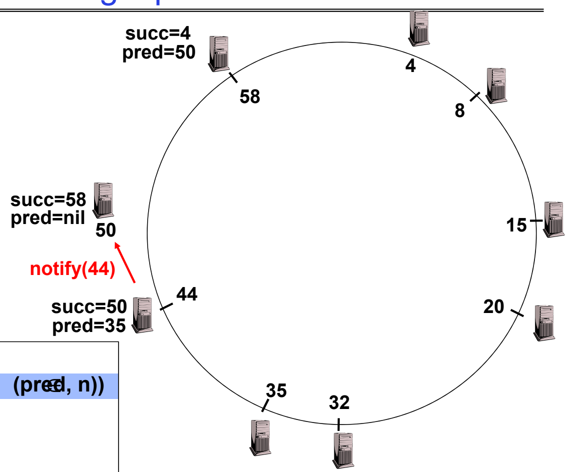
- n=44 executes stabilize()
- n=44 sends notify(44) to its successor




```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

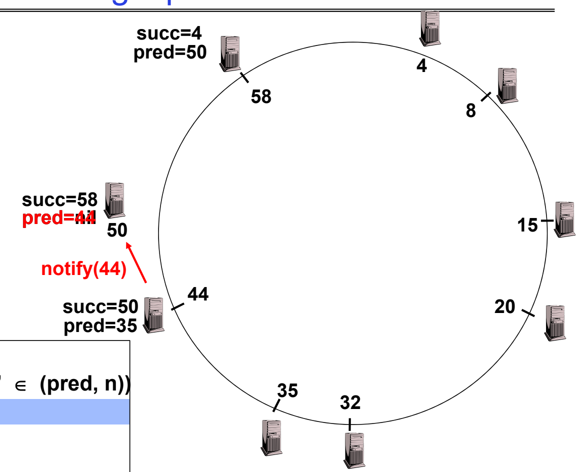
- n=50 executes notify(44)
  - pred=nil




```
n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n'
```


Joining Operation

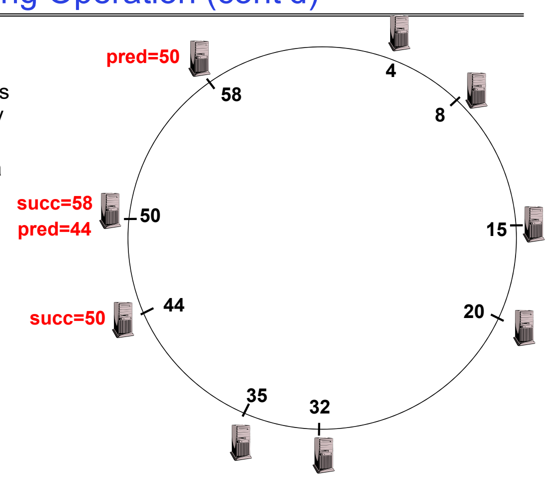
- n=50 executes notify(44)
  - pred=nil
- n=50 sets pred=44




```
n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n'
```


Joining Operation (cont'd)

- This completes the joining operation!
- The same stabilizing process will deal with failed nodes by reconnecting the ring
- What if 2 or more nodes in a row fail?
  - Keep track of more neighbors!
  - Called the "leaf set"

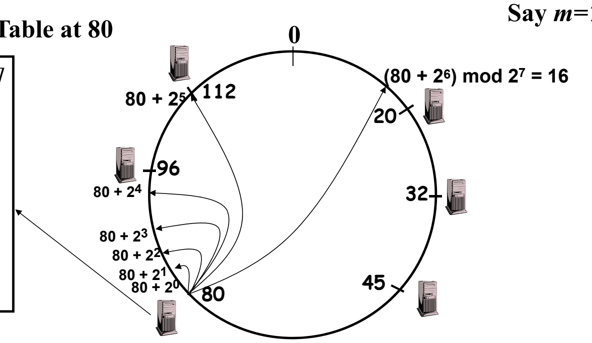


Achieving Efficiency: *finger tables*

Say *m=7*

Finger Table at 80

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



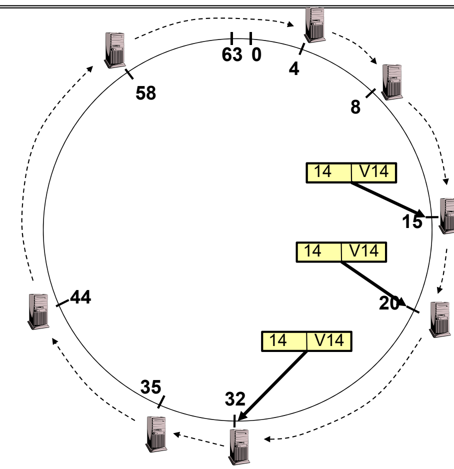
*i*th entry at peer with id *n* is first peer with id >= $n+2^i \pmod{2^m}$

Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
  - Again – called the "leaf set"
  - In the pred() reply message, node A can send its k-1 successors to its predecessor B
  - Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list
- If k = log(M), lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system

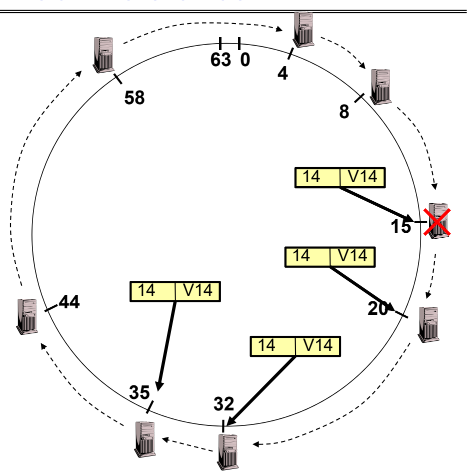
Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example: replicate (K14, V14) on nodes 20 and 32

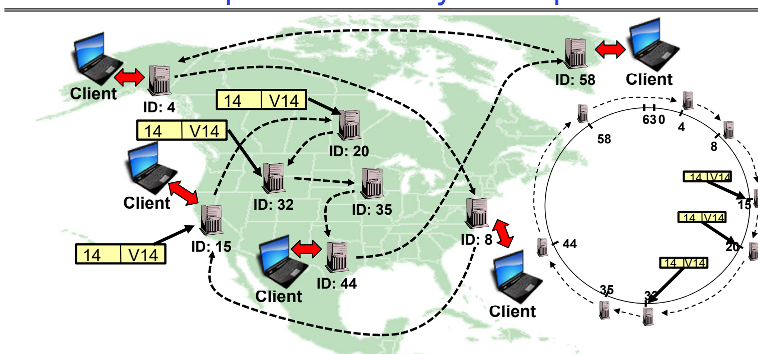


Storage Fault Tolerance

- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed after stabilization
- Will need to add a new replica on node 35



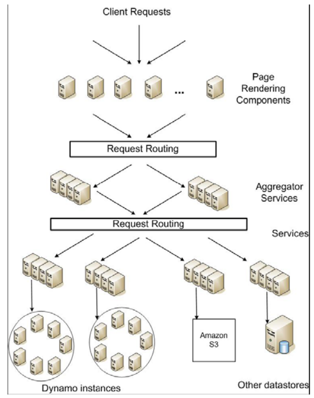
Replication in Physical Space



- Replicating in Adjacent nodes of virtual space ⇒ Geographic Separation in physical space
  - Avoids single-points of failure through randomness
  - More nodes, more replication, more geographic spread

DynamoDB Example: Service Level Agreements (SLA)

- Dynamo is Amazon's storage system using "Chord" ideas
- 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
- VFS: Virtual File System layer (Or Virtual Filesystem Switch)
  – Provides mechanism which gives same system call interface for different types of file systems
- 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

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
- 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