CS162
Operating Systems and Systems Programming
Lecture 24

Distributed Storage, Key Value Stores, Chord

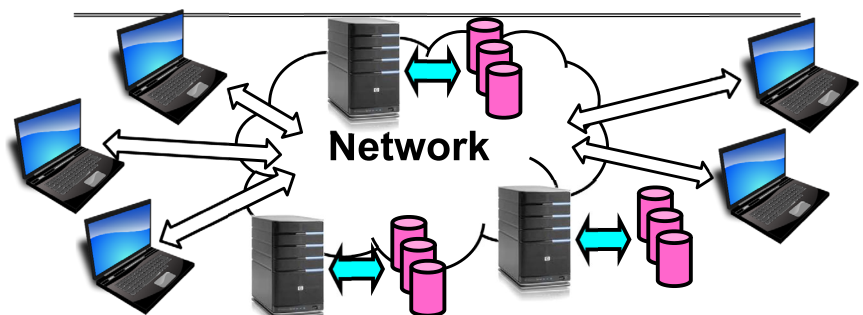
April 28th, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: TCP Congestion Avoidance

- Congestion Avoidance algorithm:
  – Pick amount of outstanding data to avoid overloading network while still maximizing pipelining
  – Assumption: most data loss caused by overloading!
- How does the sender's window size get chosen?
  – Must be less than receiver's advertised buffer size
  – Try to match the rate of sending packets with the rate that the slowest link can accommodate
  – Sender uses an adaptive algorithm to decide window size
    » Goal: fill network between sender and receiver
    » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
  – So, how much data do we want in the network baring congestion?
    » Round-trip latency × bandwidth of slowest link
    » Ack arrives just as we are about to send next packet
- TCP solution: "slow start" (start sending slowly)
  – If no timeout, slowly increase window size (throughput) by 1 for each ACK received
  – Timeout ⇒ congestion, so cut window size in half
  – "Additive Increase, Multiplicative Decrease"

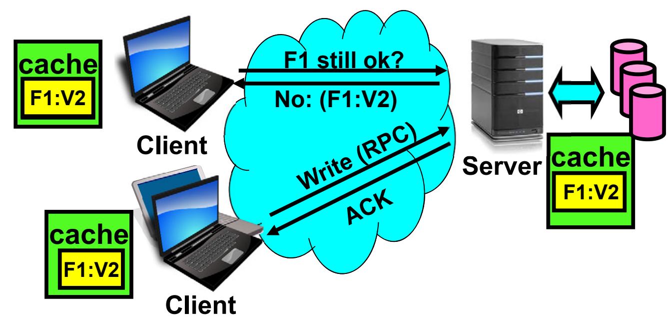
Recall: The CAP Theorem



- 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"

Recall: NFS Cache consistency

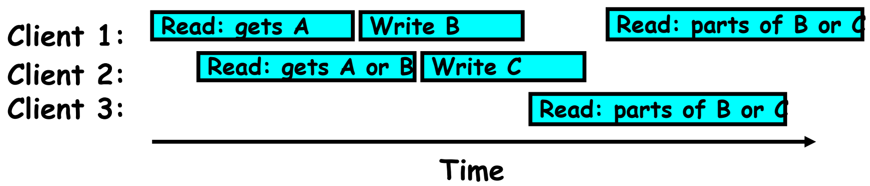
- NFS protocol: weak consistency
  – Client polls server periodically to check for changes
    » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    » Thus, when file is changed on one client, server is notified, but other clients use old version of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



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

NFS: 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

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

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

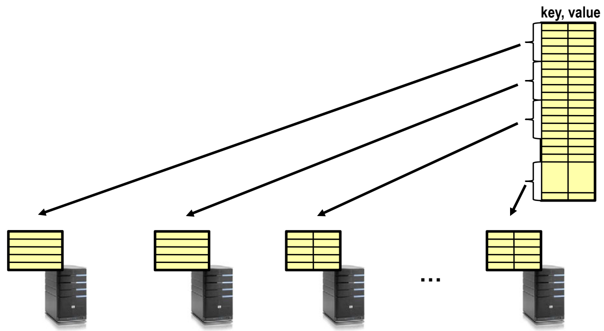
- Amazon:
  - Key: customerID
  - Value: customer p history, credit card, ..)
- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

Key-value storage systems in real life

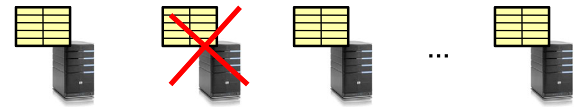
- **Amazon**
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
- **Cassandra**: "distributed data management system" (developed by Facebook)
- **Memcached:** in-memory key-value 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: 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

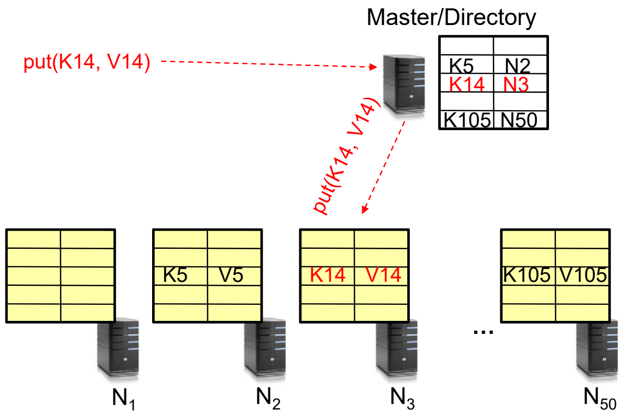
- **put(key, value)**:
  – where do you store a new (key, value) tuple?
- **get(key)**:
  – where is the value associated with a given "key" stored?
- And, do the above while providing
  – Scalability
  – Fault Tolerance
  – Consistency

How to solve the "where?"

- Hashing to map key space $\Rightarrow$ location
  – 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?
- Lookup
  – Hmm, won't this be a bottleneck and single point of failure?

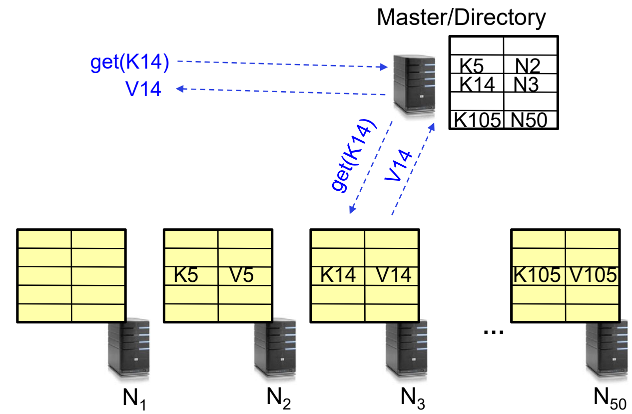
Recursive Directory Architecture (put)

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



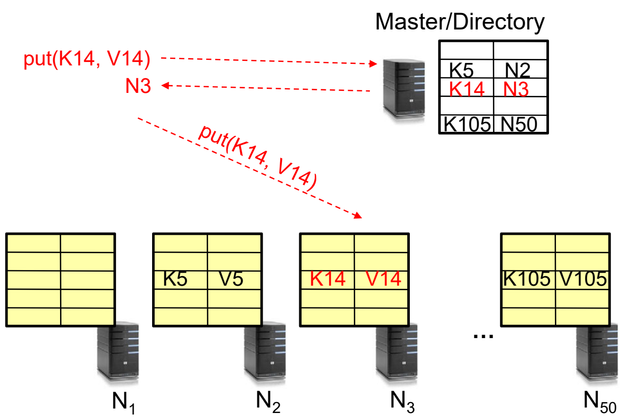
Recursive Directory Architecture (get)

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



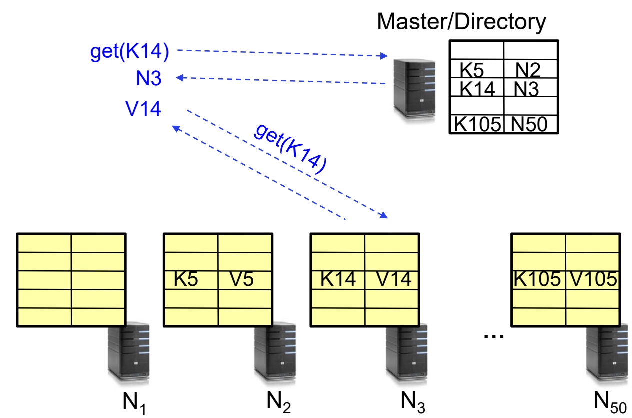
Iterative Directory Architecture (put)

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

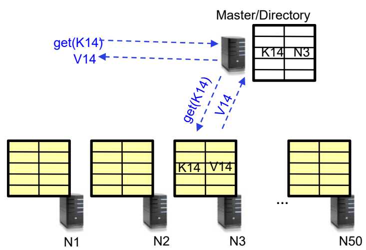


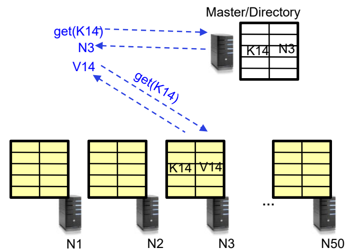
Iterative Directory Architecture (get)

- 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

\+ 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

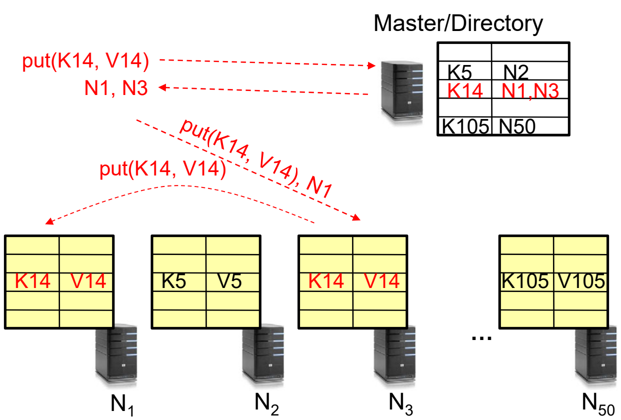
\- Harder to enforce consistency

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

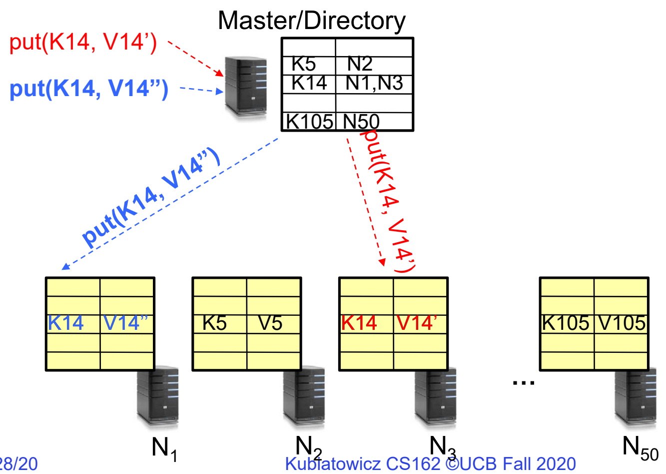


Consistency

- 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)

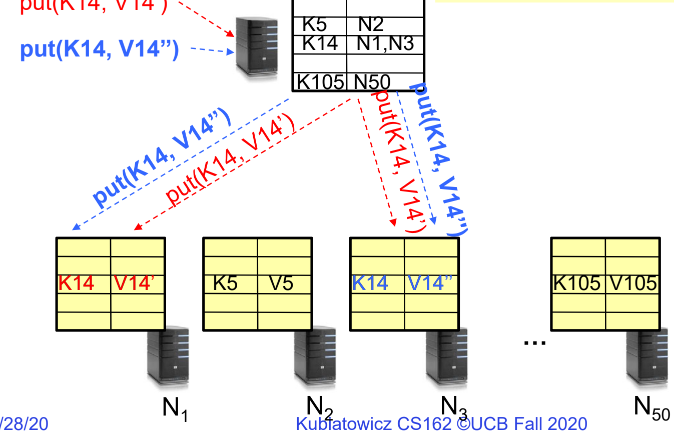
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



Consistency (cont'd)

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

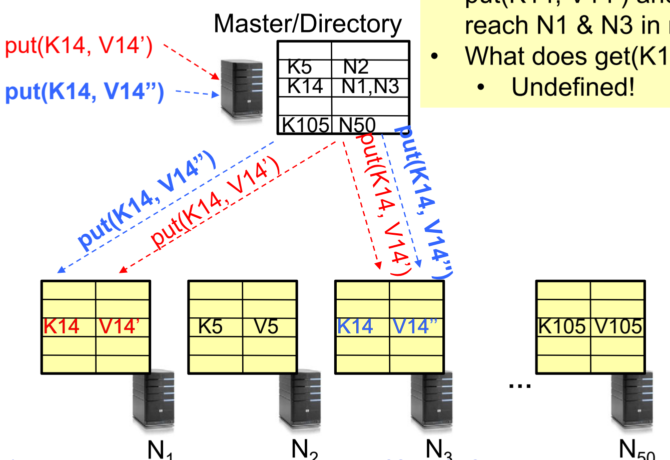
- put(K14, V14') and put(K14, V14'') reach N1 & N3 in reverse order!



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

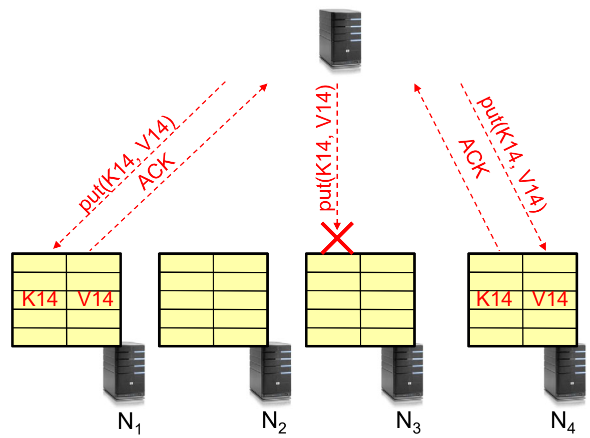
- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  – Think "one updated at a time"
  – Transactions
- Eventual consistency: given enough time all updates will propagate through the system
  – One of the 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?

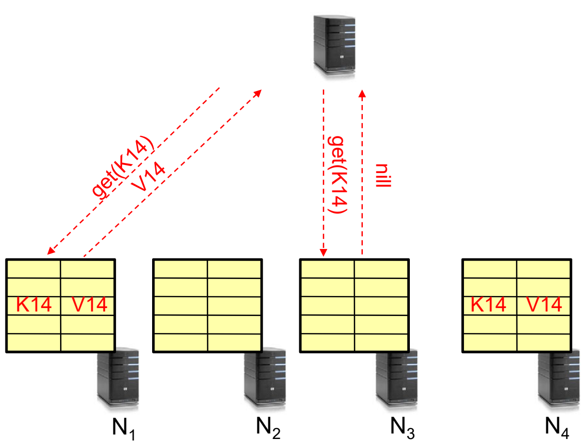
Quorum Consensus Example

- 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

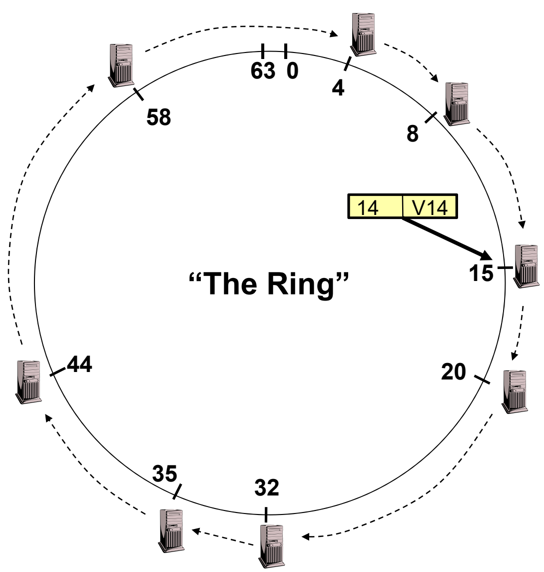
- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node
- What happens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

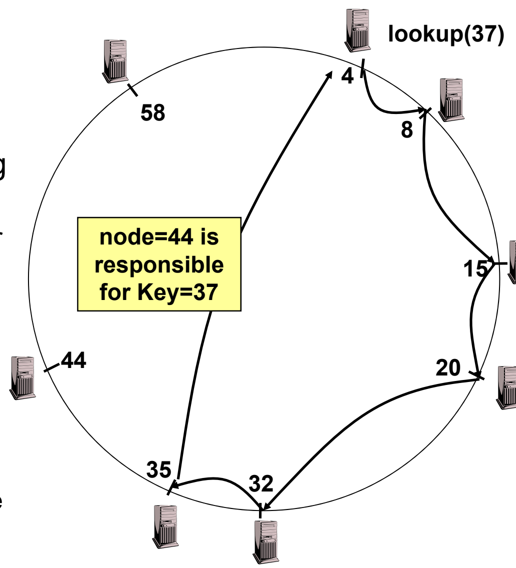


Chord: Distributed Lookup (Directory) Service

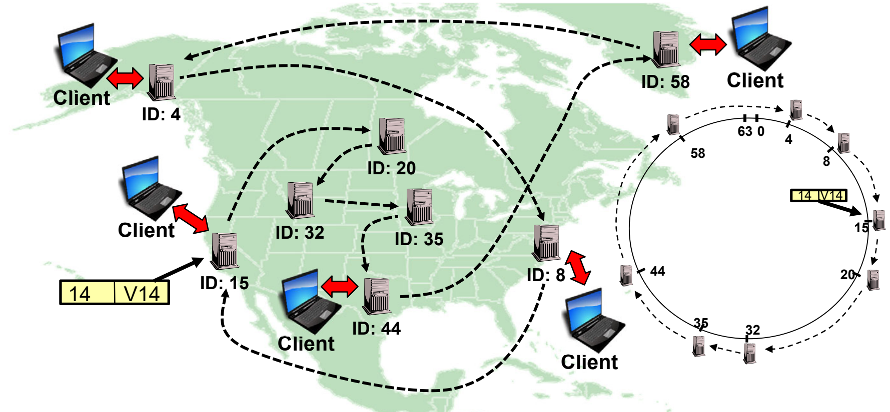
- "Chord" is a Distributed Lookup Service
  - Designed at MIT and here at Berkeley (Ion Stoica among others)
  - Simplest and cleanest algorithm for distributed storage
    - » Serves as comparison point for other optims
- Import aspect of the design space:
  - Decouple correctness from efficiency
 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

Stabilization Procedure

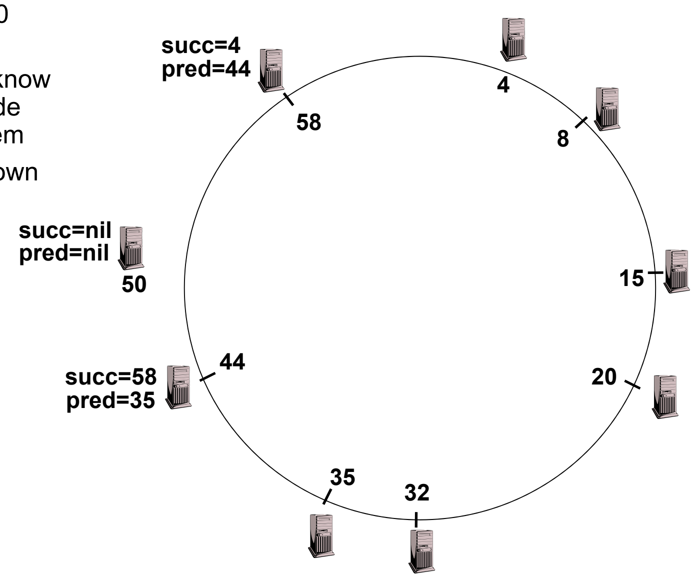
- Periodic operation performed by each node n to maintain its successor when new nodes join the system
  - The primary Correctness constraint

```
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
```

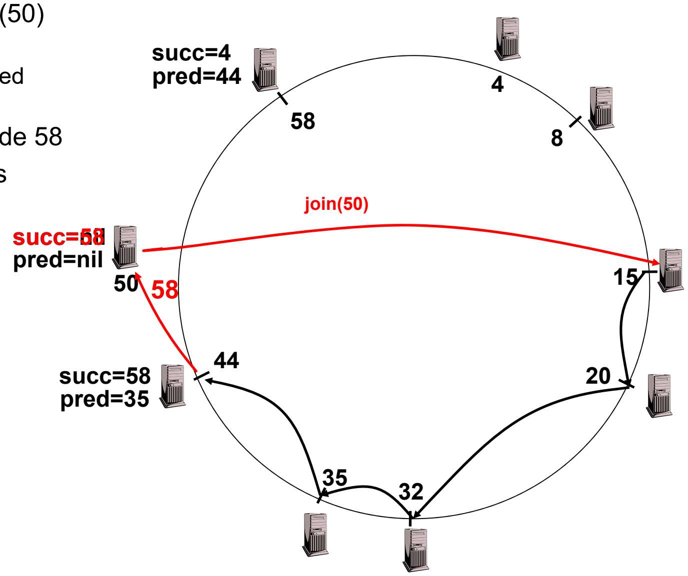

Joining Operation

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



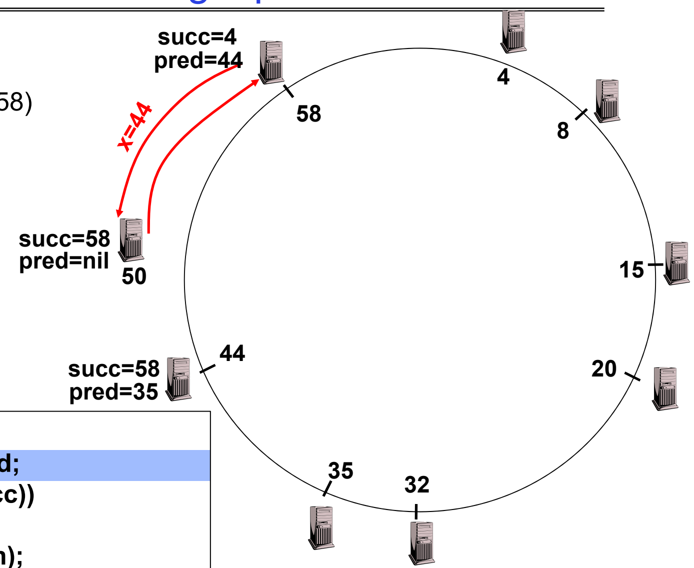
Joining Operation

- 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

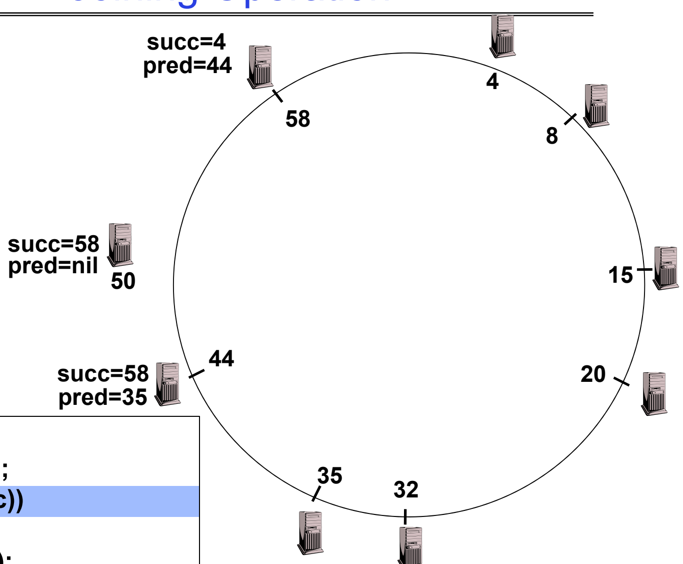
- n=50 executes stabilize()
- n's successor (58) returns x = 44




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


Joining Operation

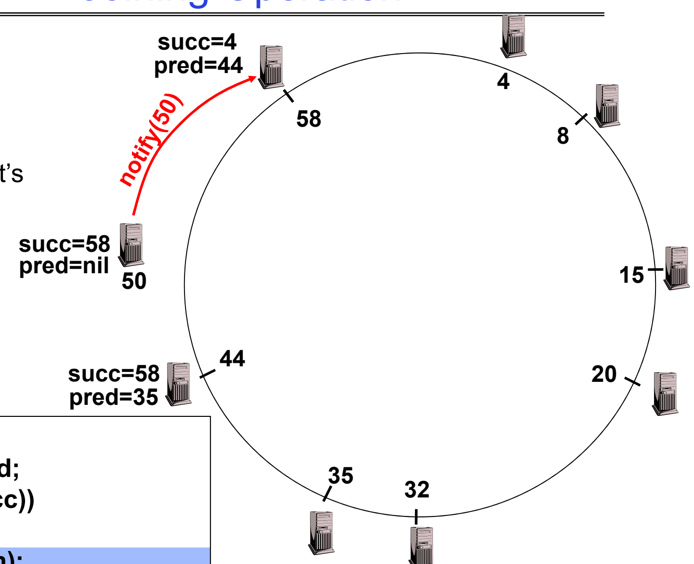
- n=50 executes stabilize()
  - x = 44
  - succ = 58




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


Joining Operation

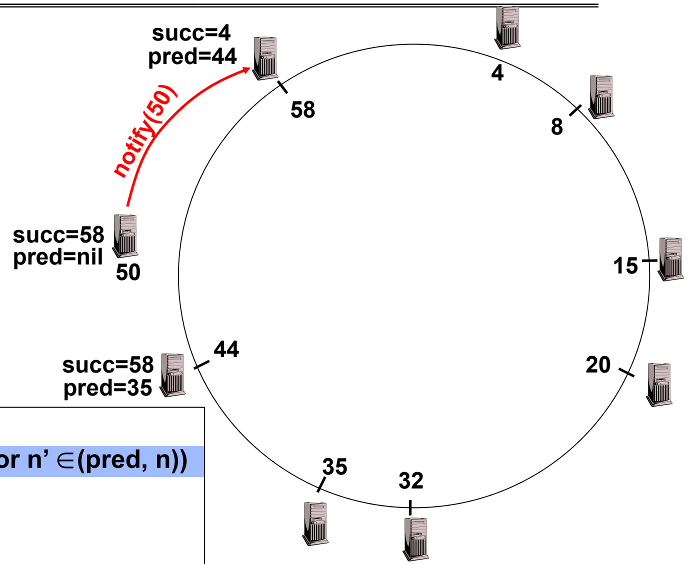
- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it's successor (58) notify(50)




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

Joining Operation

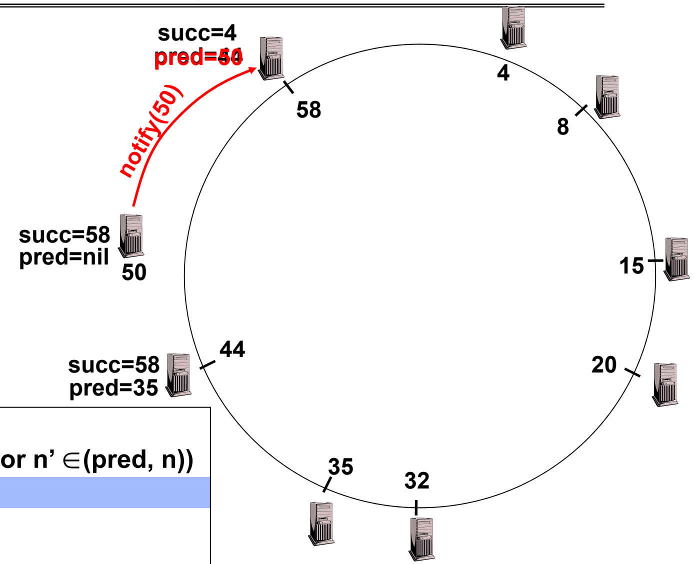
- n=58 executes notify(50)
  - pred = 44
  - n' = 50




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


Joining Operation

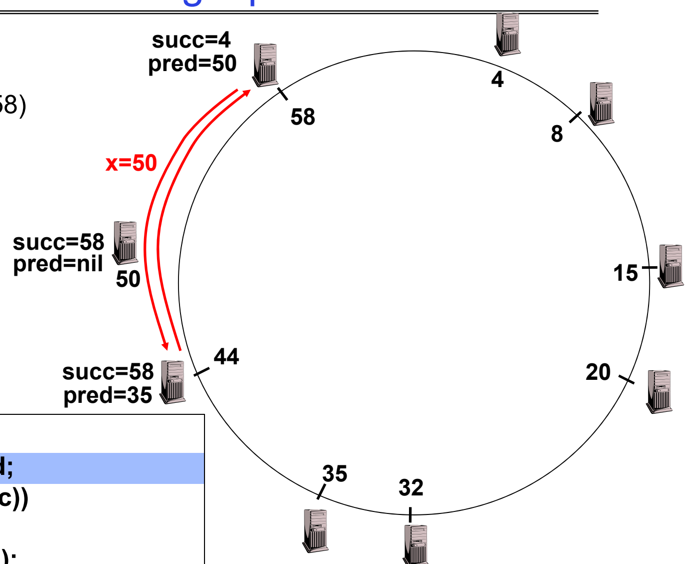
- n=58 executes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50




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


Joining Operation

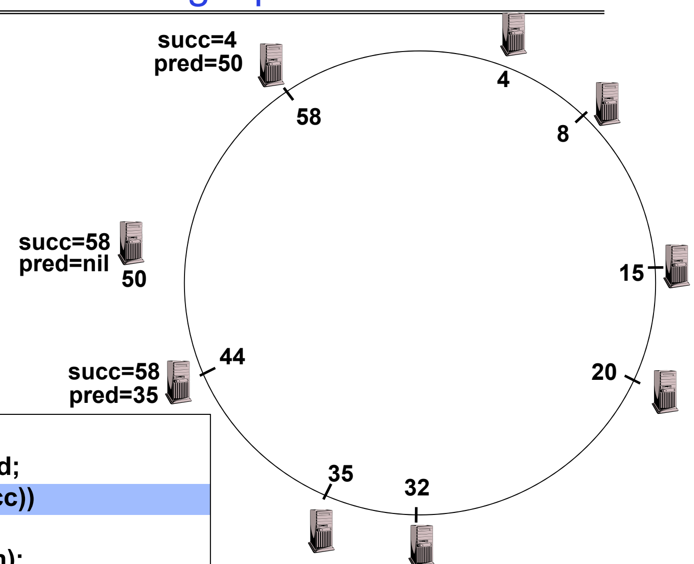
- n=44 executes stabilize()
- n's successor (58) returns x=50




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


Joining Operation

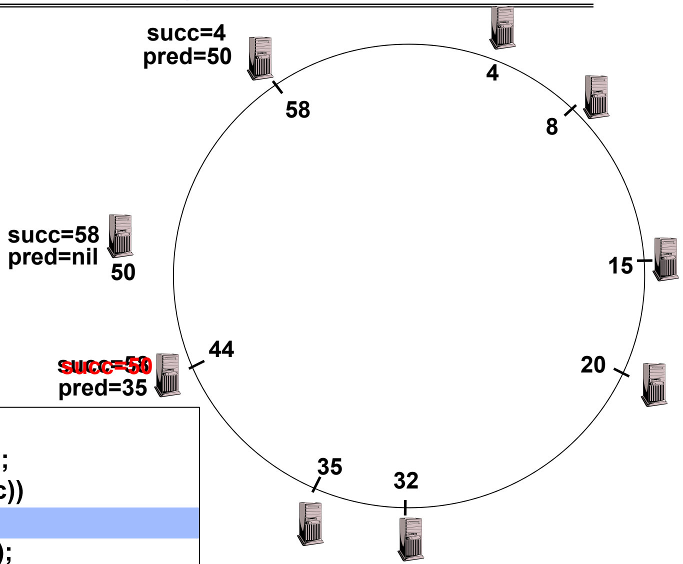
- n=44 executes stabilize()
  - x=50
  - succ=58




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

Joining Operation

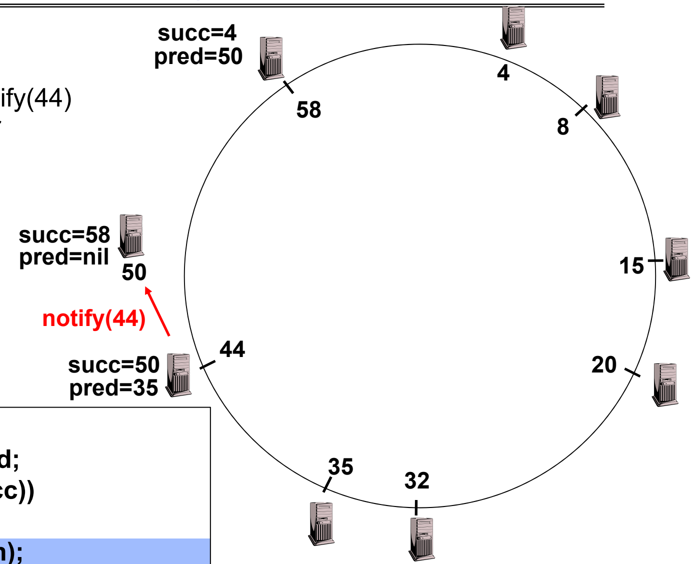
- n=44 executes stabilize()
  – x=50
  – succ=58
- n=44 sets succ=50




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


Joining Operation

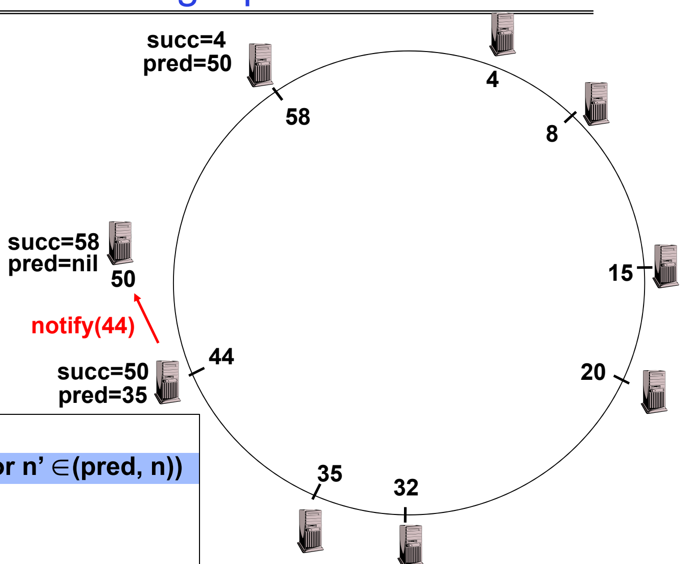
- n=44 executes stabilize()
- n=44 sends notify(44) to its successor




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


Joining Operation

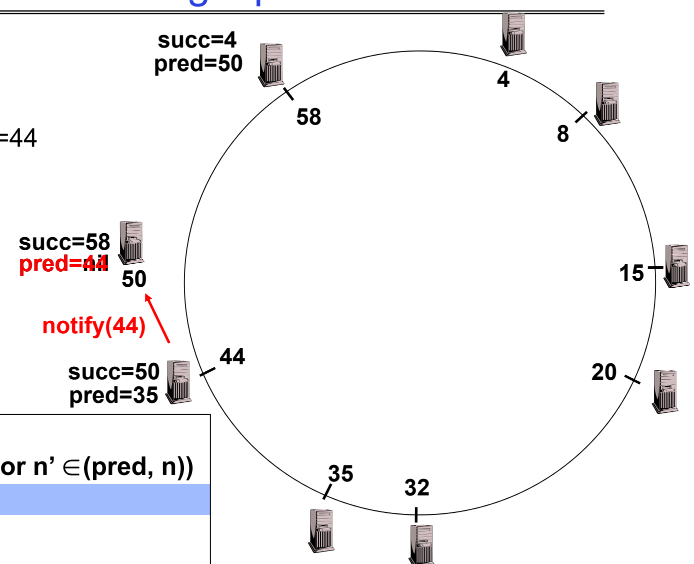
- n=50 executes notify(44)
  – pred=nil




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


Joining Operation

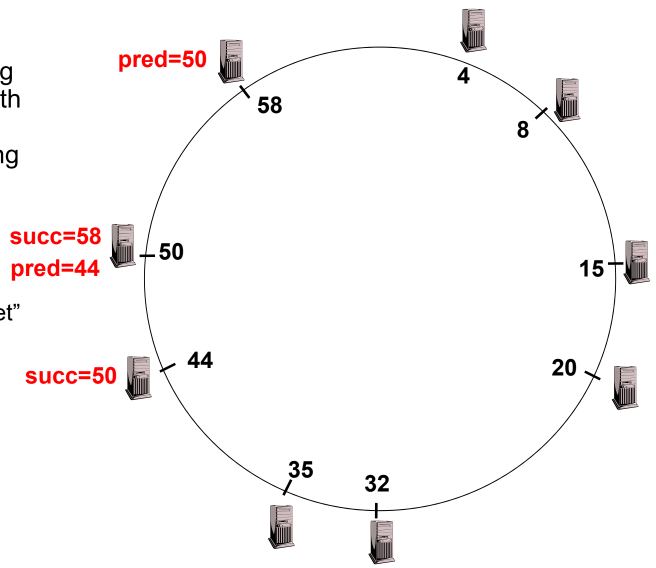
- n=50 executes notify(44)
  – pred=nil
- n=50 sets pred=44




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

Joining Operation (cont'd)

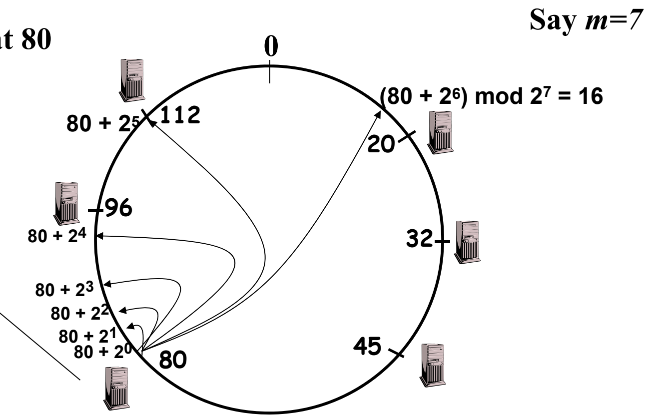
- This completes the joining operation!
- The same stabilizing process will deal with failed nodes by reconnecting the ring
- What if 2 or more nodes in a row fail?
  - Keep track of more neighbors!
  - Called the "leaf set"



Achieving Efficiency: *finger tables*

Finger Table at 80

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



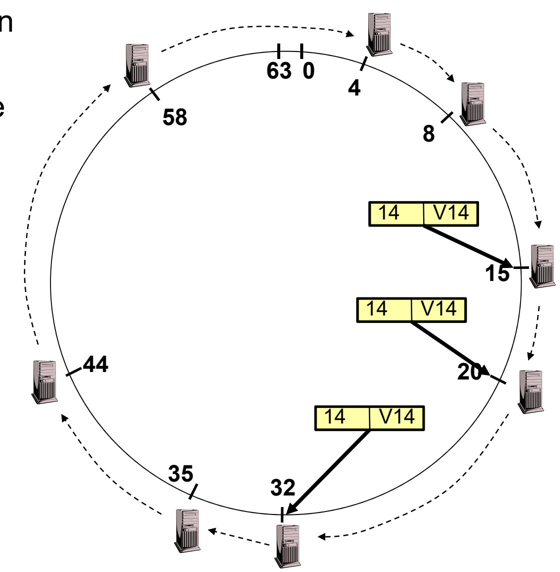
*i*th entry at peer with id *n* is first peer with id >= $n+2^i \pmod{2^m}$

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

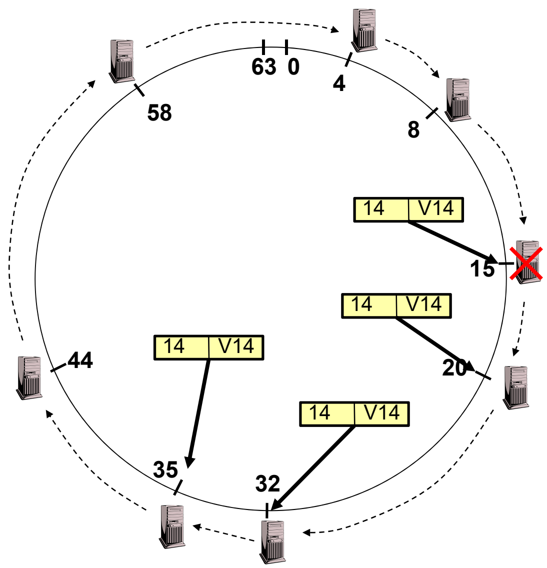
Storage Fault Tolerance

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

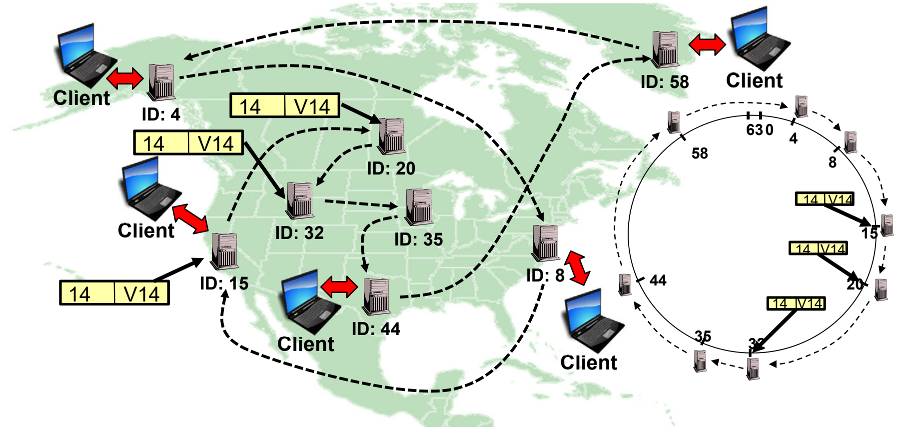


Storage Fault Tolerance

- 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



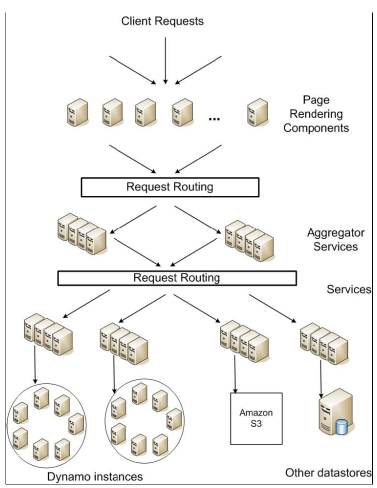
Replication in Physical Space



- 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

Extra Slides (If time!): Security

- Not to worry about for Midterm 3!

What is Computer Security Today?

- Computing in the presence of an adversary!
  - Adversary is the security field's defining characteristic
- Reliability, robustness, and fault tolerance
  - Dealing with Mother Nature (random failures)
- Security
  - Dealing with actions of a knowledgeable attacker dedicated to causing harm
  - Surviving malice, and not just mischance
- Wherever there is an adversary, there is a computer security problem!

CIMPLICITY®

BlackEnergy SCADA malware (Supervisory Control and Data Acquisition)

Mirai IoT botnet

Protection vs. Security

- Protection: mechanisms for controlling access of programs, processes, or users to resources
  - Page table mechanism
  - Round-robin schedule
  - Data encryption
- Security: use of protection mechanisms to prevent misuse of resources
  - Misuse defined with respect to policy
    - » E.g.: prevent exposure of certain sensitive information
    - » E.g.: prevent unauthorized modification/deletion of data
  - Need to consider external operational environment
    - » Most well-constructed system cannot protect information if user accidentally reveals password – social engineering challenge

On The Importance of Data Integrity

- In July (2015), a team of researchers took total control of a Jeep SUV remotely
- They exploited a firmware update vulnerability and hijacked the vehicle over the Sprint cellular network
- They could make it speed up, slow down and even veer off the road
- Machine-to-Machine (M2M) communication has reached a dangerous tipping point
  - Cyber Physical Systems use models and behaviors that from elsewhere
  - Firmware, safety protocols, navigation systems, recommendations, …
  - IoT (whatever it is) is everywhere
- Do you know where your data came from? *PROVENANCE*
- Do you know that it is ordered properly? *INTEGRITY*
- *The rise of Fake Data!*
  - *Much worse than Fake News…*
  - *Corrupt the data, make the system behave very badly*

Security Requirements

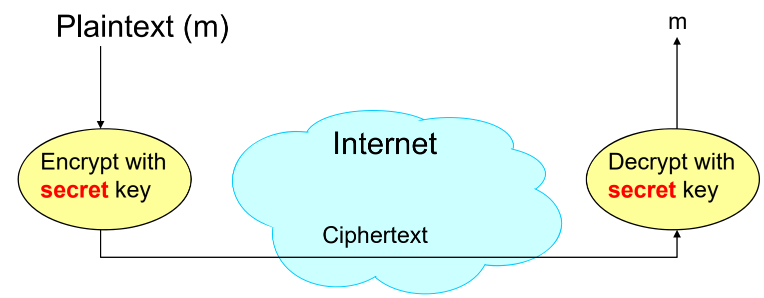
- Authentication
  - Ensures that a user is who is claiming to be
- Data integrity
  - Ensure that data is not changed from source to destination or after being written on a storage device
- Confidentiality
  - Ensures that data is read only by authorized users
- Non-repudiation
  - Sender/client can't later claim didn't send/write data
  - Receiver/server can't claim didn't receive/write data

Securing Communication: Cryptography

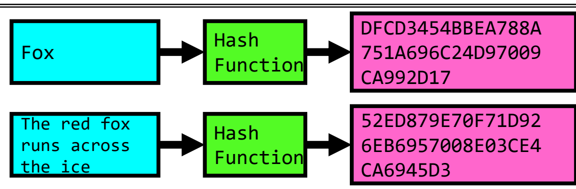
- Cryptography: communication in the presence of adversaries
- Studied for thousands of years
  - See the Simon Singh's The Code Book for an excellent, highly readable history
- Central goal: confidentiality
  - How to encode information so that an adversary can't extract it, but a friend can
- General premise: there is a key, possession of which allows decoding, but without which decoding is infeasible
  - Thus, key must be kept secret and not guessable

Basic Tool: Using Symmetric Keys

- Same key for encryption and decryption
- Achieves confidentiality
- Vulnerable to tampering and replay attacks

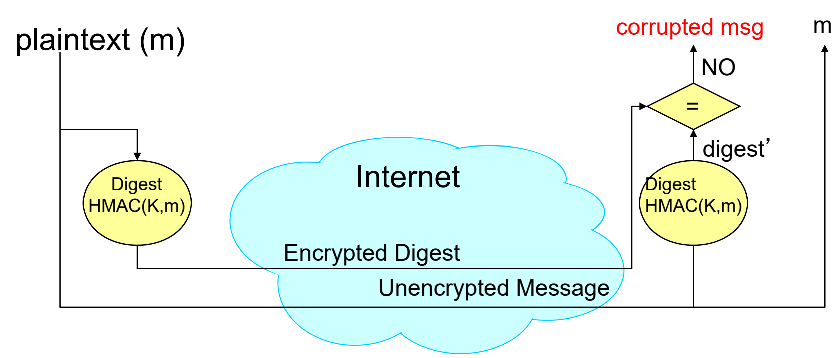


Basic Tool: Secure Hash Function



- Hash Function: Short summary of data (message)
  - For instance, $h_1=H(M_1)$ is the hash of message M_1
    - » h_1 fixed length, despite size of message M_1
    - » Often, h_1 is called the "digest" of M_1
- Hash function H is considered secure if
  - It is infeasible to find M_2 with $h_1=H(M_2)$; i.e., can't easily find other message with same digest as given message
  - It is infeasible to locate two messages, m_1 and m_2, which "collide", i.e. for which $H(m_1) = H(m_2)$
  - A small change in a message changes many bits of digest/can't tell anything about message given its hash

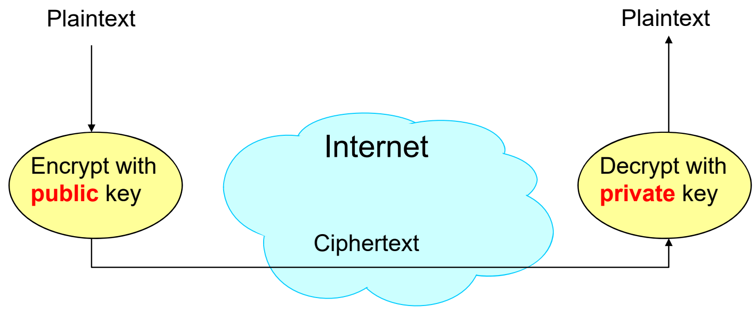
Using Hashing for Integrity



Can encrypt m for confidentiality

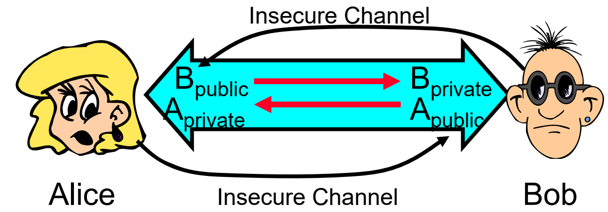
Basic Tool: Public Key / Asymmetric Encryption

- Sender uses receiver's public key
  - Advertised to everyone
- Receiver uses complementary private key
  - Must be kept secret



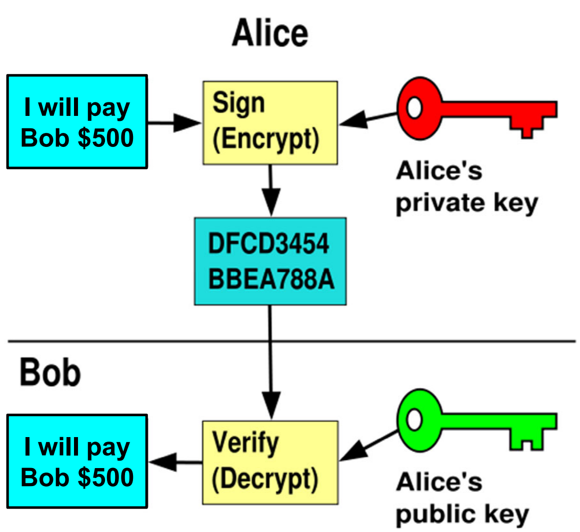
Public Key Encryption Details

- Idea: K_{public} can be made public, keep $K_{private}$ private

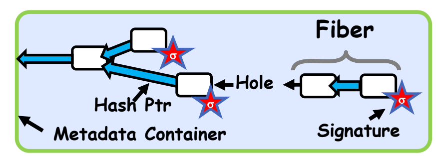


- Gives message privacy (restricted receiver):
  - Public keys (secure destination points) can be acquired by anyone/used by anyone
  - Only person with private key can decrypt message
- What about authentication?
  - Use combination of private and public key
  - Alice→Bob: $[(\text{I'm Alice})^{Aprivate}\ \text{Rest of message}]^{Bpublic}$
  - Provides restricted sender and receiver
- But: how does Alice know that it was Bob who sent her B_{public}? And vice versa… Story for another time!

Public Key Crypto & Signatures



The Data-Centric Vision: Cryptographically Hardened Data Containers

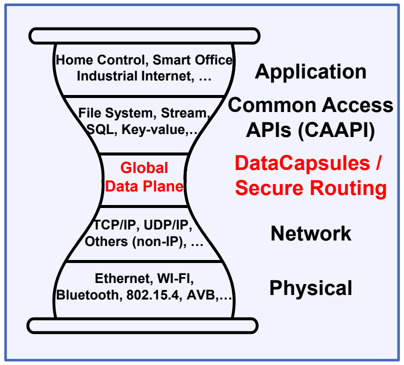


- Inspiration: Shipping Containers
  - Invented in 1956. Changed everything!
  - Ships, trains, trucks, cranes handle standardized format containers
  - Each container has a unique ID
  - Can ship (and store) anything
- Can we use this idea to help?

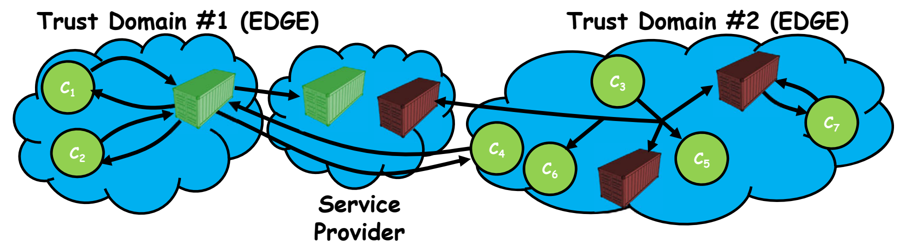
- DataCapsule (DC):
  - Standardized metadata wrapped around opaque data transactions
  - Uniquely named and globally findable
  - Every transaction explicitly sequenced in a hash-chain history
  - Provenance enforced through signatures
- Underlying infrastructure assists and improves performance
  - Anyone can verify validity, membership, and sequencing of transactions (like blockchain)

Refactoring of Applications around Security, Integrity, and Provenance of Information

- Goal: A thin Standardized entity that can be easily adopted and have immediate impact
  - Can be embedded in edge environments
  - Can be exploited in the cloud
  - Natural adjunct to Secure Enclaves for computation
- DataCapsules ⇒ bottom-half of a blockchain?
  - Or a GIT-style version history
  - Simplest mode: a secure log of information
  - Universal unique name ⇒ permanent reference
- Applications writers think in terms of traditional storage access patterns:
  - File Systems, Data Bases, Key-Value stores
  - Called Common Access APIs (CAAPIs)
  - DataCapsules are always the Ground Truth



Global Data Plane (GDP) and the Secure Datagram Routing Protocol



- Flat Address Space Routing
  - Route queries to DCs by names, independent of location (e.g. no IP)
  - DCs move, network deals with it
  - Short-term Channels ("μ-SSL channels")
- Black Hole Elimination
  - Only servers authorized by owner of DC may advertise DC service
- Routing only through domains you trust!
  - Secure Delegated Flat Address Routing
- Secure Multicast Protocol
  - Only clients/DC storage servers with proper (delegation) certificates may join
- Queries (messages) are Fibers
  - Self-verifying chunks of DataCapsules
  - Writes include appropriate credentials
  - Reads include proofs of membership
- Incremental deployment as an overlay
  - Prototype tunneling protocol ("GDPinUDP")
  - Federated infrastructure w/routing certificates

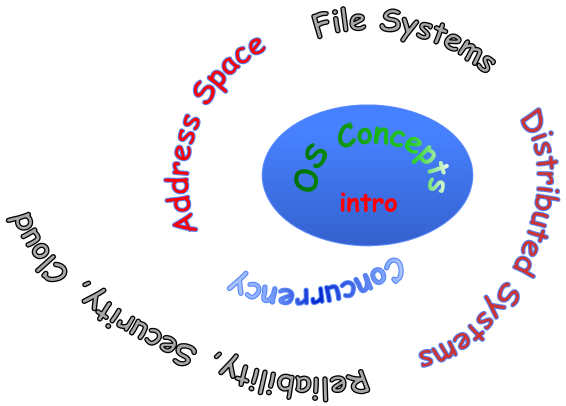
Summary (1/2)

- Distributed File System:
  - Transparent access to files stored on a remote disk
  - Caching for performance
- VFS: Virtual File System layer
  - 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

Thank you!



- Thanks for all your great questions!
- Good Bye!