Recall: Network Layering

- **Layering:** building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size"
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

Physical Reality: Packets  
Abstraction: Messages

<table>
<thead>
<tr>
<th>Physical Reality: Packets</th>
<th>Abstraction: Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Reliable</td>
<td>Unreliable</td>
</tr>
<tr>
<td>Machine-to-machine</td>
<td>Process-to-process</td>
</tr>
<tr>
<td>Only on local area net</td>
<td>Routed anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Insecure</td>
<td>Secure</td>
</tr>
</tbody>
</table>

Recall: IPv4 Packet Format

- **IP Packet Format:**
  - IP Header Length
  - Size of datagram (header+data)
  - Flags & Fragmentation to split large messages
  - 16-bit identification
  - TTL
  - Protocol
  - 16-bit header checksum
  - 32-bit source IP address
  - 32-bit destination IP address
  - Data

<table>
<thead>
<tr>
<th>IP Ver4</th>
<th>IP Header Length</th>
<th>Size of datagram (header+data)</th>
<th>Flags &amp; Fragmentation to split large messages</th>
<th>16-bit identification</th>
<th>TTL</th>
<th>Protocol</th>
<th>16-bit header checksum</th>
<th>32-bit source IP address</th>
<th>32-bit destination IP address</th>
<th>Data</th>
</tr>
</thead>
</table>

- **IP Protocol field:**
  - 8 bits, distinguishes protocols such as TCP, UDP, ICMP
- **IP Datagram:** an unreliable, unordered, packet sent from source to destination
  - Function of network – deliver datagrams!

Recall: Internet Transport Protocols

- **Datagram service (UDP):** IP Protocol 17
  - No-frills extension of “best-effort” IP
  - Multiplexing/Demultiplexing among processes
- **Reliable, in-order delivery (TCP):** IP Protocol 6
  - Connection set-up & tear-down
  - Discarding corrupted packets (segments)
  - Retransmission of lost packets (segments)
  - Flow control
  - Congestion control
  - More on these in a moment!
- Other examples:
  - DCCP (33), *Datagram Congestion Control Protocol*
  - RDP (26), *Reliable Data Protocol*
  - SCTP (132), *Stream Control Transmission Protocol*
- Services not available
  - Delay and/or bandwidth guarantees
  - Sessions that survive change-of-IP-address
  - Security/denial of service resilience/…
**Example: UDP Transport Protocol**

- The Unreliable Datagram Protocol (UDP)
  - Layered on top of basic IP (IP Protocol 17)
  - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)

  - UDP adds minimal header to deliver from process to process (i.e. the source and destination Ports)
  - Important aspect: low overhead!
    - Often used for high-bandwidth video streams
    - Many uses of UDP considered “anti-social” – none of the “well-behaved” aspects of (say) TCP/IP

**UDP Data**

- 16-bit UDP length
- 16-bit UDP checksum
- 16-bit source port
- 16-bit destination port

**IP Header**

(20 bytes)

```
IP Header (20 bytes)
```

```
16-bit source port 16-bit destination port
16-bit UDP length 16-bit UDP checksum
```

**UDP Data**

**Application Layer (7 - not 5!)**

- **Service**: any service provided to the end user
- **Interface**: depends on the application
- **Protocol**: depends on the application

  - Examples: Skype, SMTP (email), HTTP (Web), Halo, BitTorrent …

  - What happened to layers 5 & 6?
    - “Session” and “Presentation” layers
      - Part of OSI architecture, but not Internet architecture
      - Their functionality is provided by application layer
        - E.g. RPC is thought of as a “session” layer
        - E.g. Encoding is a “Presentation” mechanism. MIME, XDR

**Putting it all together**

```
101010100110101110
```

```
Transport Layer
```

```
Network Layer
```

```
Datalink Layer
```

```
Physical Layer
```

```
101010100110101110
```

```
Application Layer
```

```
Trans. Hdr. Data
```

```
Net. Hdr. Trans. Hdr. Data
```

```
```

```
101010100110101110
```

```
Physical Layer
```

```
Application Layer
```

```
Trans. Hdr. Data
```

```
Net. Hdr. Trans. Hdr. Data
```

```
```

```
101010100110101110
```

```
Physical Layer
```

**Five Layers Summary**

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
- Logically, layers interacts with peer’s corresponding layer

```
Application
```

```
Transport
```

```
Network
```

```
Datalink
```

```
Physical
```

```
Host A
```

```
Router
```

```
Host B
```

4/25/19 Kubiatowicz CS162 ©UCB Fall 2019 Lec 23.7
Physical Communication

- Communication goes down to physical network
- Then from network peer to peer
- Then up to relevant layer

Network Details: sk_buff structure

- Socket Buffers: sk_buff structure
  - The I/O buffers of sockets are lists of sk_buff
    » Pointers to such structures usually called “skb”
  - Complex structures with lots of manipulation routines
  - Packet is linked list of sk_buff structures
Avoiding Interrupts: NAPI

- NAPI ("New API"): Use polling to receive packets
  - Only some drivers actually implement this
- Exit hard interrupt context as quickly as possible
  - Do housekeeping and free up sent packets
  - Schedule soft interrupt for further actions
- Soft Interrupts: Handles reception and delivery

Using Acknowledgements

- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ACK") when packet received properly at destination
  - Timeout at sender: if no ACK, retransmit
- Some questions:
  - If the sender doesn’t get an ACK, does that mean the receiver didn’t get the original message?
    - No
  - What if ACK gets dropped? Or if message gets delayed?
    - Sender doesn’t get ACK, retransmits, Receiver gets message twice, ACK each

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - If transmit close to maximum rate, get more throughput – even if some packets get lost
    - If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - Point-to-point network: insufficient queue at switch/router
    - Broadcast link: two hosts try to use same link
    - In any network: insufficient buffer space at destination
    - Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery on top of Unreliable Packets
  - Need to make sure that packets actually make it to receiver
    - Every packet received at least once
    - Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

How to Deal with Message Duplication?

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate number’s; Discard if detected
- Requirements:
  - Sender keeps copy of unACK’d messages
    - Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don’t send next message until ACK received
  - Sender keeps last message; receiver tracks sequence number of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency \times throughput)
  - Con: doesn’t work if network can delay or duplicate messages arbitrarily
Better Messaging: Window-based Acknowledgements

- Windowing protocol (not quite TCP):
  - Send up to N packets without ack
    » Allows pipelining of packets
    » Window size (N) < queue at destination
  - Each packet has sequence number
    » Receiver acknowledges each packet
    » ACK says "received all packets up to sequence number X"/send more

- ACKs serve dual purpose:
  - Reliability: Confirming packet received
  - Ordering: Packets can be reordered at destination

- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ACK packet
  - Receiver gets packets out of order!
    » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    » Reduces # of retransmits, but more complex

- What if ACK gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets

Transmission Control Protocol (TCP)

- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)

- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    » "Window" reflects storage at receiver – sender shouldn’t overrun receiver’s buffer space
    » Also, window should reflect speed/capacity of network – sender shouldn’t overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    » A "good citizen"

TCP Windows and Sequence Numbers

- Sender has three regions:
  - Sequence regions
    » sent and ACK’d
    » sent and not ACK’d
    » not yet sent
  - Window (colored region) adjusted by sender

- Receiver has three regions:
  - Sequence regions
    » received and ACK’d (given to application)
    » received and buffered
    » not yet received (or discarded because out of order)

Administrivia

- Last Midterm: 5/2
  - Can have 3 handwritten sheets of notes – both sides
  - Focus on material from lecture 17-24, but all topics fair game!

- Midterm Time is now: 5-7PM
  - It is earlier, during class period (+30 minutes)
  - Please let us know if you conflict situation changed
    » Watch Piazza for room assignments

- Please come to class on 4/30!
  - HKN evaluations!

- Don’t forget to do your group evaluations!
  - Very important to help us understand your group dynamics
  - Important to do this for Project 3 as well!
    » Even though it will be after Midterm 3!
Window-Based Acknowledgements (TCP)

- **Congestion**
  - How long should timeout be for re-sending messages?
    - Too long wastes time if message lost
    - Too short: retransmit even though ACK will arrive shortly
  - Stability problem: more congestion => ACK is delayed => unnecessary timeout => more traffic => more congestion
    - Closely related to window size at sender: too big means putting too much data into network
- **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 size of N
    - Goal: fill network between sender and receiver
    - Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- **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”

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:** local name: 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?
      - 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 resends 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 is a tunable parameter).
  - Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.

**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”
  - Client 1: Read: gets A, Write B, Read: parts of B or C
  - Client 2: Read: gets A or B, Write C
  - Client 3: Read: parts of B or C

**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

- System-calls interface
  - VFS interface
    - UNIX file system
    - NFS client
    - NFS server
  - Other types of file systems

Enabling Factor: Virtual Filesystem (VFS)

- 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”
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.

Linux VFS

- An operations object is contained within each primary object type to set operations of specific filesystems
  - "super_operations": methods that kernel can invoke on a specific filesystem, i.e. write_inode() and sync_fs().
  - "inode_operations": methods that kernel can invoke on a specific file, such as create() and link().
  - "dentry_operations": methods that kernel can invoke on a specific directory entry, such as d_compare() or d_delete().
  - "file_operations": methods that process can invoke on an open file, such as read() and write().
- There are a lot of operations!

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

Important Questions

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

- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency
Directory-Based Architecture (1/4)

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

Directory-Based Architecture (2/4)

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

Directory-Based Architecture (3/4)

- 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 (4/4)

- Having the master relay the requests → recursive query
- Another method: iterative query (this slide)
  - 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 (1/3)

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

Fault Tolerance (2/3)

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

Fault Tolerance (3/3)

- 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
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

• 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, N2, N4\}
- Assume put() on N3 fails

#### 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

- Partitioning example with \(m=8\) \(\Rightarrow\) 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 optim
- 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 better successor, update
if (x \notin (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' \notin (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

```
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

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

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

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

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

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

```
n=50 executes stabilize()
  - x = succ.pred;
  - if (x < (n, succ))
    - succ = x;
    - succ.notify(n);
```

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

```
n=50 executes 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 its successor (58) notify(50)

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

• n=50 executes stabilize()
  - pred = 44
  - n’ = 50
- set pred = 50

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

Joining Operation

- n=58 executes notify(50)
  - pred = 44
  - n’ = 50
- n’s successor (58) returns x=50

n.notify(n’)
if (pred = nil or n’ ∈ (pred, n))
  pred = x;
succ.notify(n);

• 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);

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

n=44 executes stabilize()
  x = succ.pred;
  if (x ⊂ (n, succ))
    succ = x;
    succ.notify(n);

notify(44)

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

n.notify(n')
  if (pred = nil or n' ⊂ (pred, n))
    pred = n'
**Joining Operation**

- \(n=50\) executes \(\text{notify}(44)\)
  - \(\text{pred}=\text{nil}\)
- \(n=50\) sets \(\text{pred}=44\)

**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**

**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 \(\text{pred}()\) reply message, node A can send its \(k-1\) successors to its predecessor B
  - Upon receiving \(\text{pred}()\) 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

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 ⇒ Geographic Separation
  - 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
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