Internet Computing

Giant Scale Services

```
* Giant Scale Services Introduction
    - How do you program big data applications, like search engines, to
    run on large scale clusters?
    - Aim of this lesson is to discuss scalability to thousands of processes
    and tolerating failures
* Giant Scale Services
    - Online airline reservation
    - Purchasing a laptop online
    - Web mail
    - Google search
    - Streaming a movie from Netflix
* Tablet Introduction
    - Systems issues in giant scale services
    - Programming models for applications working on big data
    - Content distribution networks
* Generic Service Model of Giant Scale Services
    - Typical architecture for giant scale services:
        + Many servers connected through high-bandwidth communication
        backplane
        + Load manager distributing work across servers; balances client
        traffic and hide partial failures
        + Data store accessible by databases
        + IP network communicates to load manager (traffic is generally
        independent, "embarrassingly parallelizable")
* Clusters as Workhorses
    - Statistics below from E. Brewer's paper "Lessons from Giant-Scale
    Services" circa 2000
    - 10x to 100x scale today
    - Advantage of using clusters is that hardware can easily be added as
    demand increases
```

Service	Nodes	Queries	Nodes
AOL Web Cache	>1000	10B/day	4 CPU DEC 4100s
Inktomi Search Engine	>1000	>80M/day	2-CPU Sun Workstations
Geocities	>300	>25M/day	PC Based
Anonymous Web-based e-mail	>5000	>1B/day	FreeBSD PCs

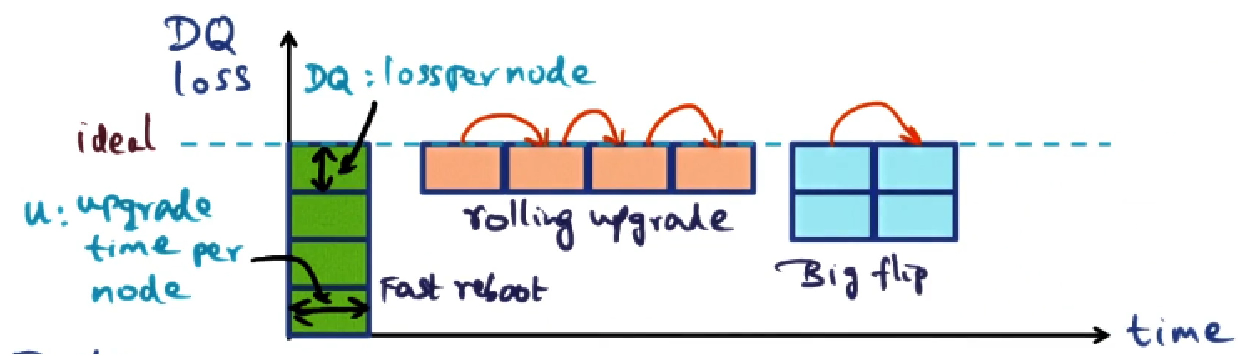
```
* Load Management Choices
    - OSI Reference Model
        1. Application
        2. Presentation
        3. Session
        4. Transport
        5. Network
        6. Link
        7. Physical
    - Higher layers allow for increasing functionality of the load manager
* Load Management at Network Level
    - Round robin Domain Name Server (DNS): All requests have the same
    domain name to all come to this service "google.com"
```

        + DNS directs incoming requests to separate IP addresses within the cluster
    - All servers identical
    - Replicated data for each server
    - Pros:
        + Good load balancing
    - Cons:
        + Cannot hide down server nodes from external world
* Load Management at Transport Level (or higher)
    - Layer 4 switches to direct traffic to servers
    - Provides opportunity to dynamically isolate down server nodes from the external world
    - Can direct requests based on their type (search, mail, game)
    - Still provides good load balancing
    - Data can be partitioned among all servers
        + Servers communicate to serve queries
        + Can be data loss if a server is down (typically replicate in addition to partition)
* DQ Principle
    - Server has all the data for handling incoming client requests (Df)
        + Df: All data available to servers
        + Dv: Available data for processing queries (due to failures, etc.)
        + Harvest: D = Dv/Df (ideally 1)
    - Clients issues requests with their queries (Qo)
        + Qo: Offered load by clients
        + Qc: Completed requests by servers
        + Yield: Q = Qc/Qo (ideally 1)
    - DQ is a constant for a given server capacity
        + Can increase Q by decreasing D and vice versa (inversely related)
        + Use less data and process more requests
        + Harvest/yield tradeoff
    - Bound by network capacity (can add more servers)
    - If a server fails, we can either decrease Q or D
        + Depends on how cluster is configured
    - Server measurement: I/O operations/second
    - Giant scale operations are network bound, not I/O bound, so DQ is a better metric
    - Quantifying uptime
        + Mean time between failures (MTBF)
        + Mean time to repair (MTTR)
        + Uptime = (MTBF - MTTR) / MTBF
        + Want uptime as close to 1 as possible
        + If there are no queries during MTTR, then there's no issue, so uptime isn't the most intuitive measure of how a server is performing; harvest and yield are preferable
* Replication vs Partitioning
    - Replicated: Every data server has full corpus of data needed to service a particular request
        + When a node fails, D is unchanged and Q decreases
    - Partitioned: Each server only contains a portion of the data
        + When a node fails, D decreases and Q is unchanged
    - DQ is independent of replicated vs partitioned data (constant)
        + Disk space is cheap so processing incoming requests is network bound, not disk bound

+ If a request requires a large amount of disk I/O, replication may be more expensive than partitioning

- Beyond a certain point, replication is likely preferable because failures are inevitable. Therefore, some of the data is unavailable in a partitioned scheme.
    + Users will prefer having all of their data for an email service
    + It might be okay to not have all the data in a search service

* Graceful Degradation
  - If a server is saturated (reached the DQ limit) then we can degrade the server from the point of view of the client
    + D constant, Q decreased (same fidelity to fewer clients)
    + D decreased, Q constant (worse fidelity to all clients)
  - DQ provides an explicit strategy for managing saturation
* Online Evolution and Growth
  - Services are continually evolving, so servers need to be upgraded with new software, new nodes added, or old nodes replaced with new nodes
  - While the upgrade is happening, loss of service is inevitable
  - Options for upgrading:
    + Fast reboot: Bring down all servers at once, upgrade, and bring back up (diurnal server property, do it during periods of low use)
    + Rolling upgrade: Bring down one server down at a time, upgrade, bring back up and go to the next (service is always available, but never fully available)
    + Big flip: Bring down half the nodes at once, then the other half (server capacity decreased by 50% for the entire period)
  - DQ loss for all three strategies
    + DQ * U * N
    + DQ loss for individual node
    + U: Upgrade time for individual node
    + N: Number of nodes
  - Maintainence and upgrades are controlled failures that can be handled by system administrators/developers



Upgrade and Growth of Clusters

* Giant Scale Services Conclusion
  - Giant scale services are network bound and not disk I/O bound
  - DQ principle helps system designer in creating explicit policies for graceful degradation and optimizing for yield or harvest

MapReduce

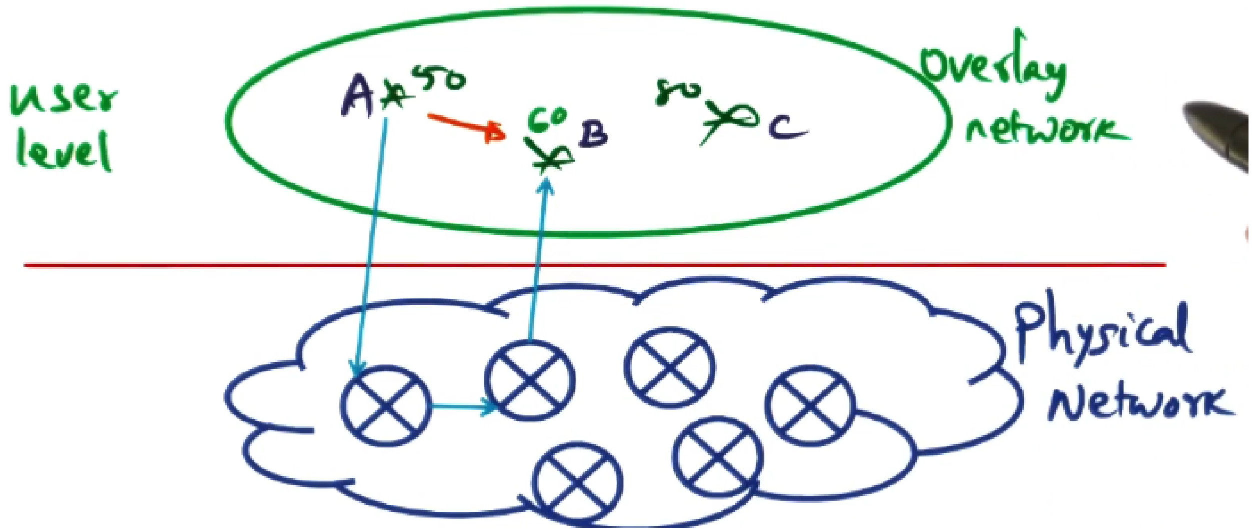
* MapReduce Introduction
    - Programming paradigm for exploiting the computational resources of the thousands of nodes in a data center
        + Example: Searching for a photo
        + Embarrassingly parallelizable: Not much management across parallel portions of the computation
        + Must be concerned with the inevitable failures of hardware
* MapReduce
    - Programmer defines two functions, map and reduce
        + Both take user-defined <key,value> pairs as input and output
    - Example: Count number of occurrence of names in documents
        + key: filename, value: contents (<f1,contents>, <f2,contents>, etc)
    - Map: Look for unique names
        + Returns key:unique name, value:number
        + Can spawn as many instances as the number of unique files
    - Reduce: Aggregates all of the outputs from the map functions
        + Output of mapper is input to reducer
        + Returns key:unique name, value:total number
    - Programmer needs to determine format of input, intermediate, and output key/value pairs, map/reduce functionality
        + No need to to worry about how many instances of mappers/reducers and plumbing between output of mappers and input of reducers
* Why MapReduce
    - Several processing steps in giant-scale services are expressible as map-reduce
        + Airline bookings, word indexes for document searches, ranking of pages when a user does a search, etc
        + Embarrassingly parallel, work with big data
    - Domain expert writes map and reduce functions
    - PageRank example
        + <url,content> -> map -> <target,url>
        + <target,url> -> reduce -> <target,source_list>
        + Ranks for pages target1 to targetn
        + Ranking target web pages by number of source pages that contain references to the target
    - Runtime does the rest
        + Instantiating number of mappers and reducers
        + Data movement
* Heavy Lifting Done by the Runtime
    - Programming library splits input library into m splits
        + m can be specified by user or automatically determined
    - Master oversees the operation, keeps track of when workers are done with their task
        + Assigns m worker threads as mappers
        + Assigns R worker threads as reducers (decided by application; i.e. number of unique names in the name counting example)
        + Plumbs mappers to reducers
    - Map phase (work done by worker thread)
        + Read local disk
        + Parse
        + Call map (intermediate key,values are buffered in memory)
        + Intermediate files on local disks (R by each mapper)

  - When a mapper finishes, it alerts the master
    + Master waits for all mappers to finish before starting reducers
  - Reduce phase (work done by worker thread)
    + Remote read from disks of all m mappers
    + Sort (organize same keys together)
    + Call reduce (supplied by user)
    + Each reduce function writes to final output files
  - M + R > N is a possible case
    + The master's job is to organize the work across the available resources to carry out the work that needs to be done
* Issues to be Handled by the Runtime
  - Master data structures
    + Location of files created by completed mappers
    + Scoreboard of mapper/reducer assignment
    + Fault tolerance
      1. Start new instances if no timely response
      2. Completion message from redundant stragglers
    + Locality management
    + Task granularity: Number of nodes available may be less than M+R
    + Backup tasks
  - User can define partitioning function over default hashing function
    + Can also specify special ordering of keys
    + Can define combining/partial ordering to increase granularity of tasks
  - Map and reduce functions must be idempotent (if performed multiple times, has no further effect)
* MapReduce Conclusion
  - Power of MapReduce is its simplicity; heavy lifting is managed by the runtime system under the covers

Content Delivery Networks

* Content Delivery Networks Introduction
  - How is the content of the Internet stored and distributed?
* Content Delivery Networks
  - Who started content distribution networks and why?
    + Napster for music sharing
* Distributed Hash Table (DHT)
  - How to locate data
    + <key,value> pair
    + Key: Content hash (Unique name generated for content)
    + Value: Node where content is stored
  - Where do you store the key-value pair so anybody can discover it?
    + Using a central server isn't scalable
    + Genesis of the idea of a distributed hash table
  - DHT: key ~ node_id for storing <key,value>
    + Content hash might be 149, node might be 150
    + Go to node 150, which tells you the data is contained at node 80
* Distributed Hash Table Details
  - Use SHA-1 to generate a unique signature for a particular content
    + SHA-1 is 160 bits
  - Name Spaces
    + Key-space: Unique keys for content
    + Node-space: Unique node-id

```
    - Objective
        + <key> -> node-id <N> such that <key> ~ <N>
        + Ideally <key> == <N>, but close enough is good enough
    - APIs
        + void putkey(key, value)
        + value getkey(key)
* CDN (An Overlay Network)
    - Once you know the node-id, how do you translate to an IP address?
    - Overlay network: A virtual network on top of the physical network
        + Build a routing table mapping node-ids to IP addresses
        + A may know how to get to B and B knows how to get to C, so even if
        A doesn't know the IP address of C, A can send it to B who will
        forward it to C
    - At the user level, a node might only be two hops away, but there may
    be many more hops on the physical network
```



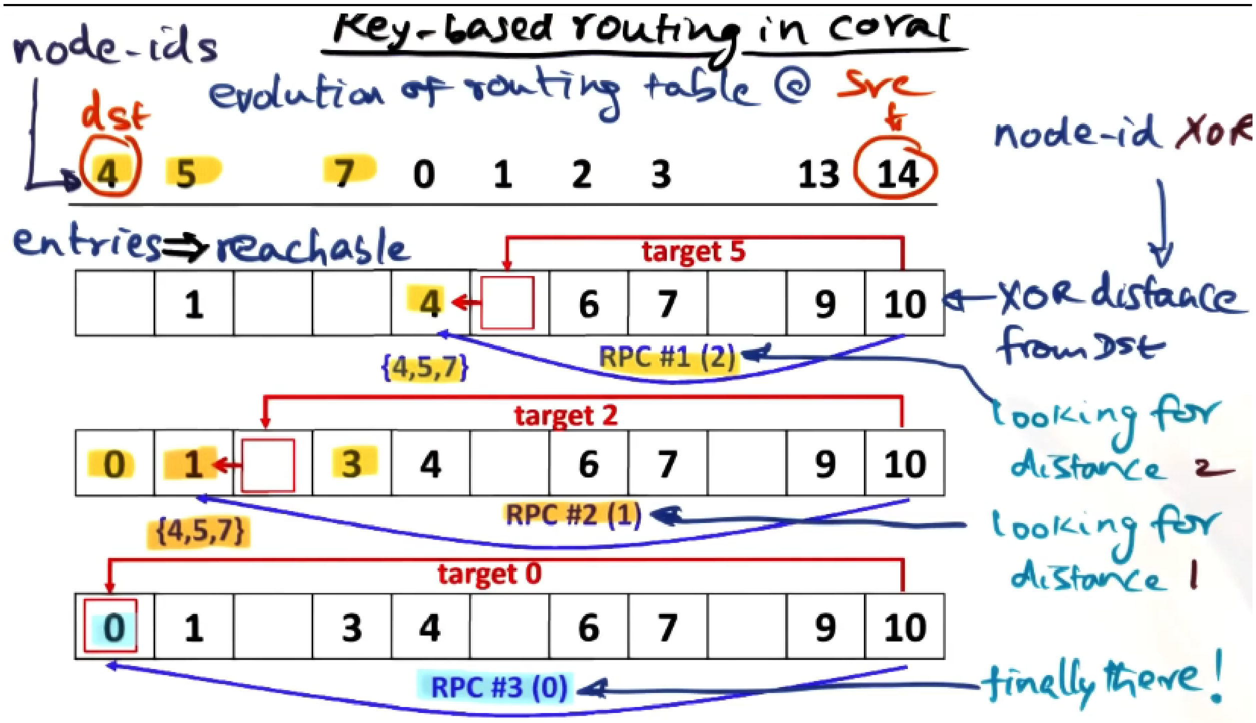
Upgrade and Growth of Clusters

```
* Overlay Networks in General
    - At the OS level, the IP network is an overlay on the LAN
        + IP address -> MAC address
    - At the app level, CDN is an overlay on TCP/IP
        + Node ID -> IP address
* Distributed Hash Tables and Content Distribution Networks
    - Placement
        + put<key, value> where key is a content hash and value is node-id
        where the content is stored
    - Retrieval of value given key
        + get<key> returns the value
* Traditional Approach
    - Place <key,value> in node N, where N ~ key
    - Retrieve:
        + Given key K go to node N (closest to key K)
    - Node space might be much bigger than local routing table
        + Routing table at each node gives every node that is reachable from
        this specific node
```

- If you want to go to a node that isn't in your local routing table, you can pick the closest node with the hope that the data will be stored in that node
    + It's possible that the content isn't stored there, so you hope that the node will know how to reach the content (another hop)
    + Repeat until you reach the content
- Attempting to reach the content in the minimum level of hops
    + At the virtual overlay network layer, not physical network layer

* Greedy Approach Leads to Metadata Server Overload
    - Four nodes generate four different puts
        + put<148,80>
        + put<149,val>
        + put<152,val>
        + put<153,val>
    - Closest node is 150, resulting in congestion at metadata server
        + Metadata server is the destination for put/get
    - If many nodes attempt to get a key simultaneously, they'll all go to the same metadata server
        + Viral video means everyone will go to the same node
    - "Tree saturation problem" where the tree is rooted at the metadata server
* Origin Server Overload
    - If a video gets popular, everybody will try to download the video from the same origin server
        + Content overload in addition to metadata overload
    - Two solutions
        1. Web proxy: Limit number of requests that go out of an organization
            + Not good enough for the slashdot effect (viral video)
        2. CDNs
            + Content mirrored and stored geographically
            + User request dynamically re-routed to geo-local mirror
            + Akamai provides CDNs to provide mirror servers; pay CDN provider to avoid origin overload
    - Coral system provides a technique for democratizing content distribution; addresses two issues
        1. Storing data in a DHT requires a scalable method that doesn't overload a metadata server
        2. Avoid origin server overload
* Greedy Approach Leads to Tree Saturation
    - Coral doesn't take the greedy approach to avoid tree saturation
    - Instead, Coral uses the K ~ N is just a hint, not absolute
    - Coral DHT
        + Get/put satisfied by nodes different from <N~key> -> "sloppy DHT"
    - Rationale
        + Avoid tree saturation
        + Spread metadata overload so no single node is saturated
    - How?
        + Novel key-based routing based on the distance between the source and destination
        + Nsrc XOR Ndst (e.g. Source=14 XOR Destination=4 = 10)
        + XOR is much faster than subtraction
        + Bigger the XOR value, larger the distance in the application namespace
* Key Based Routing

    - Valid nodes (V) are nodes that are reachable from node 14
    - Intuition in greedy approach
        + Get as close to the desired destination in node-id namespace
        + Hope he may know route to get to the desired destination
        + Only consider valid nodes from source
    - Objective: Reach destination with fewest number of hops
        + "Me first" approach leads to congestion
* Coral Key Based Routing
    - Each hop goes to some node that is half the distance to destination in node-id namespace
    - What are the nodes that are directly reachable from me? What is the XOR distance from these nodes to the destination?
    - For the following example (distance=10):
        1. Hop to 10/2 = 5
        2. Hop to 5/2 = 2
        3. Hop to 2/1 = 1
        4. Hop to destination
    - But, may not have the desired destination in the table
* Key Based Routing in Coral Example
    - Can't always reach half the distance if it isn't in your table
    - Process
        + 14 ^ 4 = 10 / 2 = 5, but no known node at 5
        + Instead, go to 4 and send RPC request for a node 5 / 2 = 2 away
        + Node 4 returns routing information for {4,5,7}
        + Want to go 2 away; doesn't exist, so go to 1 (send RPC request for a node 1 away)
        + Node 1 returns routing information for {4,5,7}
        + Looking for distance 0, so send RPC request to node 4
    - Even though the first call returned information to get to the destination, we don't skip straight there because we're not greedy
        + Still go halfway there
    - Latency to reach desired destination is increased even if you have a direct way to reach the destination
        + Placing common good ahead of own latency
        + Avoid tree saturation

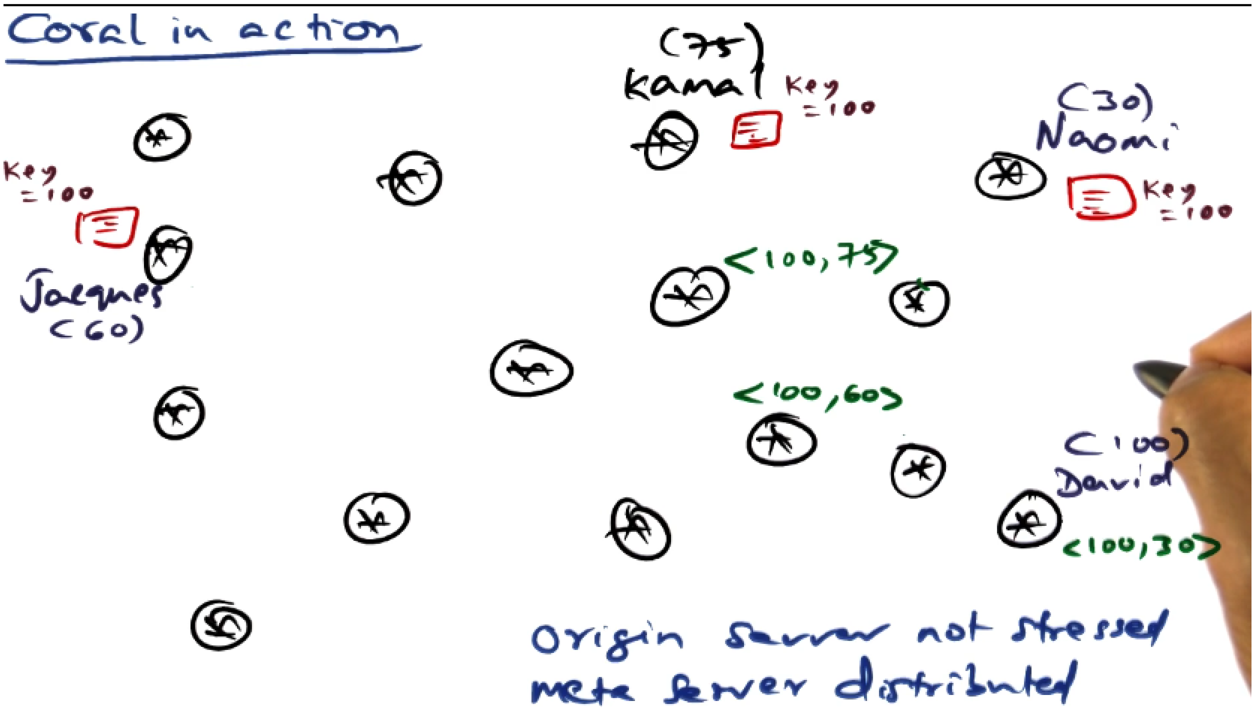


Coral Key-Based Routing

* Coral Sloppy DHT
- Primitives are the same, implementation semantics are different
- Put can be initiated by the origin server with new content or a node that just downloaded the content and wants to serve as a proxy
- put(key,value)
  + Value: node-id of proxy with content for key
  + Announcing willingness to serve as proxy
  + put<key,value> in an "appropriate" node
  + Want to put near node N for key K, but not if it's overloaded
- Determining if a node is overloaded
  + Full: max l values for key (spatial condition to entertain up to l content providers for a particular key)
  + Loaded: Max beta request rate for key (don't want many requests in a short time period)
- When putting in a key, continue going half the distance and querying if the node is full or loaded
  + If we never reach a full/loaded node, we'll reach the destination
  + If a node is full/loaded, backtrack one and insert there (assume that the following nodes are also busy)
- Forward path: Keep going forward until you reach a loaded/full node
- Retract path: Go back until you find someone willing to host your key/value pair
  + Must recheck the condition to confirm the node isn't busy due to subsequent requests
  + This process avoids the metadata overload
  + Destination metadata server doesn't exactly match the key

* Coral in Action
- Origin server not stressed

- Metadata server distributed
- Key takeaway: Even though an individual request may have slightly more latency due to more intermediate hops, Coral handles the dynamism of the modern Internet with vast amounts of data



Coral in Action

* Content Delivery Networks Conclusion
    - Coral offers a vehicle to democratize content generation, storage, and distribution through a participatory approach
    - Akamai doesn't act in this way, actually replicates mirrors