Lecture 3: Scaling by Load Balancing

1. Comments on reviews
    - i.
2. Topic 1: Scalability
    - a. QUESTION: What are problems?
        - i. These papers look at distributing load
    - b. QUESTION: What is the context?
        - i. How to build a web site or web cache to serve maximum load
        - ii. Assume static documents (this is old …)
        - iii. Assume shared back end storage (draw picture!)
    - c. QUESTION: What are concerns?
        - i. Avoid hotspots:
            1. Load on a single document can exceed capacity of a single machine
            2. Web workloads show huge variation in popularity – millions of hits per day vs none, and can exceed any server size in load.
        - ii. Leverage memory
            1. Size of documents exceeds memory size of cache
            2. Want to use aggregate capacity of all machines
        - iii. Reduce latency
            1. Avoid multi-step lookups
        - iv. Dynamic
            1. things can change – avoid moving data when they do
    - d. Question: What are simple approaches:
        - i. Round robin: distribute workload round robin to front ends
            1. Yields cache size equivalent to a single machine – all nodes hold same, popular data
        - ii. Hierarchical cache / peer-to-peer caching
            1. Ask one cache, it talks to others
        - iii. Problems:
            1. locality, making most of memory
            2. latency of talking to multiple machines
    - e. System model:
        - i. Front ends access cache, which can fetch from back ends
        - ii. Models use of Memcached by many sites
        - iii. Models web proxy caches
3. LARD
    - a. Comments from reviews:
        - i. TCP handoff only offloads one direction
            1. Forward direction packets are small (requests, acks) while reverse are large (data)
            2. Can forward packets in kernel
        - ii. Single point of failure in front end node.
            1. Solution: primary/backup – can send map of urls to backup

periodically

- iii. Use with HTTP 1.1 persistent connections
  - 1. Need to have front end parse requests & fetch objects
- iv. Are backends that can access shared disks reasonable?
- v. Concurrent updates?
  - 1. Note: LARD is for read-only data
- vi. Policy for removing nodes
  - 1. Always reduces # of replicas after 20 minutes if not growing
  - 2. What if stays high?

b. NOTE: LARD ideas are used in commercial products (layer 7 switching/LB)

c. QUESTION: What is basic system setup?

- i. Large web site with cacheable content (e.g. videos in Youtube, facebook pictures)
  - 1. Want to serve at cache rates, not disk rates
- ii. Any backend can server any data (all access common database)
  - 1. Used by Facebook for lots of things
- iii. Front end director parses requests, sends to a back end to be serviced
  - 1. Requests can queue in front end to avoid overloading back ends
- iv. Complexities:
  - 1. To parse HTTP request, need to terminate TCP connection, but want back end to send back data directly
  - 2. Want to direct requests to balance load, but keep locality for caching

d. QUESTION: What are the goals?

- i. First: maintain locality to keep highest number of different items in memory
- ii. Second: spread load to leverage all resources in system
- iii. Core problem: load imbalance. Some items are popular, other are not as popular
  - 1. Popular content 1000s of times more popular than unpopular.
  - 2. Normal hashing to distribute work or round robin leads to hot nodes with too many hits

e. Techniques:

- i. Assign URLs to nodes
  - 1. How do it?
    - a. Look at least loaded node
    - b. QUESTION: What information is needed for this?
      - i. Need centralized notion of least loaded node
    - c. QUESTION: Do you have to choose the least loaded node?
      - i. ANSWER: just pick two and choose less load
        - 1. Avoids hot spots rather than picking global best
        - 2. Can do better than picking best, because everybody picks the same best one

d. QUESTION: What if load is out of date?
   i. Picking least load has problems because you concentrate load there
   ii. Better off picking a set randomly and choosing least loaded

2. QUESTION: How do you measure load?
   a. I/O queue? Memory usage? CPU Load?
      i. Porcupine uses disk requests queue length
   b. LARD uses # of connections. Why?
      i. Info is available at front end, no need to ask for stale info from the back.
      ii. HTTP 1.0 closes connection after every request
      iii. Is good enough to work…
   c. Back-end cache policy not part of the paper.

ii. Balancing load
   1. If a node has too much load, need to spill load
   2. QUESTION: How do this?
      a. If load exceeds threshold where latency suffers, look for under loaded node
         i. Prevents idle nodes
      b. If load exceeds twice threshold, spill to any node under high threshold (even if not lightly loaded)
      c. GOAL: Prevent unnecessary spilling
   3. QUESTION: Why does this work?
      a. Evens load between low and high thresholds
   4. QUESTION: How pick Thigh – highest load before shunting data to a low-load node?
      a. ANSWER: look at response time at high load. Generally, throughput increases then flattens as load increases (becomes saturated) and latency shoots up
      b. Set Thigh to be knee in curve where latency low, but are at peak throughput
   5. SPECIFICALLY:
      a. Thigh = (Tlow + D/R)/2
         i. R = sec/req, D = secs
         ii. Thigh = (Tlow + #req from imbalance)/2
         iii. So 2*Thigh = Tlow + Req from imbalance

iii. Balancing load with replication: more than a single node of load
   1. QUESTION: How do you know the right number of servers?
      a. If load < one whole machine, the answer is 1…
      b. Overload: keep adding servers until you don't get load imbalance
      c. Underload: keep removing servers until you do get load

imbalance (but slowly)

d.

2. Start increasing number of servers if one is overloaded by picking lightly loaded node
3. Always change the set – either keep increasing or stop and decrease
    a. Prevents sending a now-dead URL to somewhere
4. Comparision: TCP/IP
    a. Additive increase to slowly ramp up
    b. Multiplicative decrease to slow down (quickly avoid congestion)
5. Here: slowly move # of nodes up and down
6. NOTE: General technique to slowly adapt to load – increase until get it right, decrease until pain
7. NOTE: Front end is managing cache sizes in the backend, but knows nothing about the caches
    a. QUESTION: Why does this work?
    b. Feedback: # of connections relates to efficiency of back end
    c. Slow back ends have connections migrated off, fast ones get more
    d. Handling things fast means connections are open shorter time, have fewer active connections at once
        i. Littles law: # of active connections = arrival rate * service time

iv. TCP Handoff

1. Front end accepts TCP connection, gets HTTP request, parses URL
2. Then packages up TCP state and sends to back end
    a. Remembers in kernel to forward packets from flow to back end
    b. Back end replies directly to client (IP spoofing)
3. Note: after redirect, just need to forward packets, no other packet inspection (can be made fast)

f. What about recovery – what if front-end nodes fail?

i. Can send load anywhere & rebuild map

g. What about state?

i. Front end must maintain a table the size of URLs to do lookup

h. What about dynamic content?

i. Dynamic content gets generated from some back end static content

ii. Can send requests to location of the underlying static content

iii. Could do dynamic generation in front end and pull underlying static content using LARD

i. Results:

i. Does better than hashing content, because it can avoid hot spots and idle nodes

- ii. Does better than weighting (WRR), because it has locality so gets better use of caches
- j. QUESTION: What are the big take-away ideas?
  - i. Request placement for maximizing cache locality
  - ii. Load balancing by evening loads (minimize difference between low/high)
  - iii. Proxying to forward requests to best backend

4. Consistent hashing
   - a. Notes from reviews:
     - i. DNS servers had to be modified to run dns helper to populate server map.
     - ii. Does DNS name caching in the client help? Could time out
     - iii. Could this be used for server-side caching?
     - iv. Is dynamic content a problem?
       1. Note: even dynamic pages have hundreds of static elements – javascript, images, style sheets
     - v. Use case: should all requests go to cache?
       1. Note: built into Akamai. Model is a web site offloads specific objects to the cache
     - vi. How deal with heterogeneous caches?
     - vii. Can it handle persistent connections?
     - viii. Is the “proximity” metric useful? Does geographic distance mean network distance?
       1. Past work shows that distance to your local DNS correlates strongly with distance to the machine
       2. Can determine what to use by what client’s dns server finds
          - a. Akamai system can return server location closest to client’s dns
     - ix. Caching can increase latency
       1. Always true: have to check cache first, so on a miss it hurts
       2. Hence want to make it fast
       3. Hence may not want to always use cache…
     - x. Could cache be further from the server than the client is?
       1. Depends on where cache is deployed.
       2. Akamai deployed caches in ISPs, close to the backbone, so tend to be closer to everything else than the client (typically)
     - xi. Issue: how do you maintain the tree?
       1. Can each server compute it locally (yes if hash of server identity)
       2. Or broadcast to servers (if servers pick locations)
     - xii. What about bad hash functions?
       1. Can use MD5 which is pretty random
     - xiii. Range queries?
       1. Do we need them? Only doing key/value lookups
     - xiv. Malicious clients going to wrong server?

1. Interesting point. How could you handle it?

b. Problem solved: building a distributed cache

i. QUESTION: What are the goals for the cache?

1. Goal 1: reduce load on backbone (fewer requests to server)
2. Goal 2: reduce latency to client
3. Goal 3: fault tolerance – keeps working well if a cache fails

ii. Constraints:

1. allow set of cache servers to change
2. Partition objects among cache servers

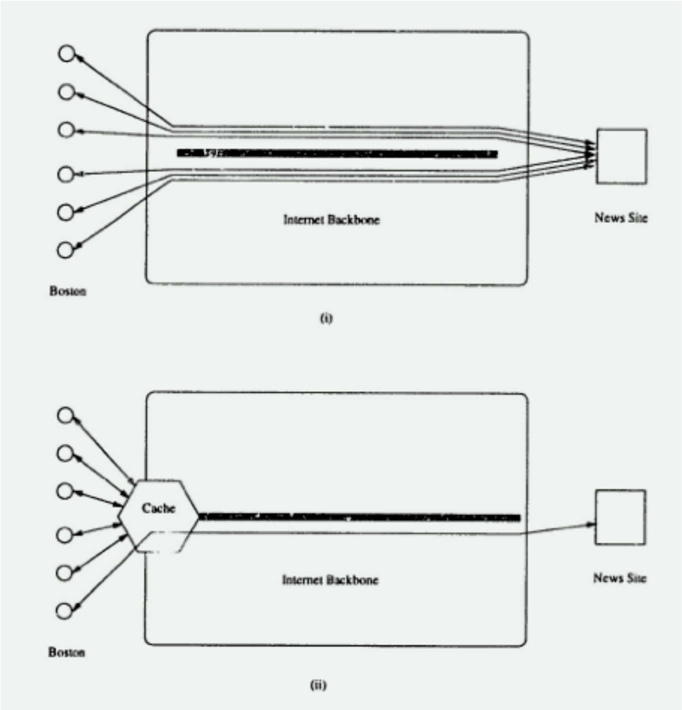
iii. NOTE: only ½ the content is cacheable

1. But many web pages have many objects (pictures, video, flash), and those binary objects are cacheable
2. ½ is still a lot to reduce.

iv. Invalidation: HTTP includes time-to-live fields, controls cacheability (for browser cache as well)

v. Model: Any cache node can fetch and cache any web page

vi. How do you get locality so that you use the memory capacity of a cache?



vii.

viii. NOTE: get scalability, because different sets of clients have a unique set of caches. This is client-side caching, not server-side caching (Different from LARD)

ix.

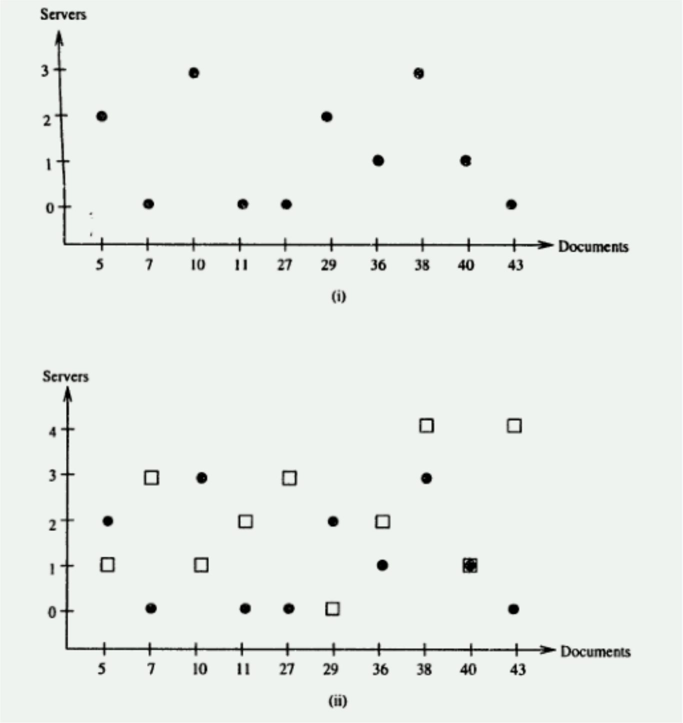
c. Constraints:

i. Want to keep latency low, so now referrals/redirections/remote communication

ii. Want to partition data so maximally use cache

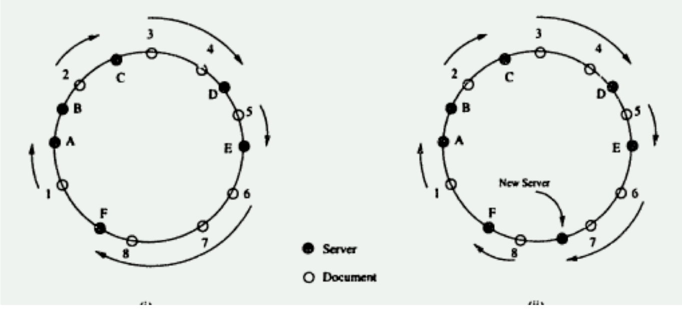
d. QUESTION: What are some possible approaches?
    i. Cooperative caching / P2P: send request to one cache, it asks others if it does not have it
    ii. Hierarchical caching: send to cache, it asks its parent, etc.
    iii. Broadcast: send to all caches, the one with the data responds

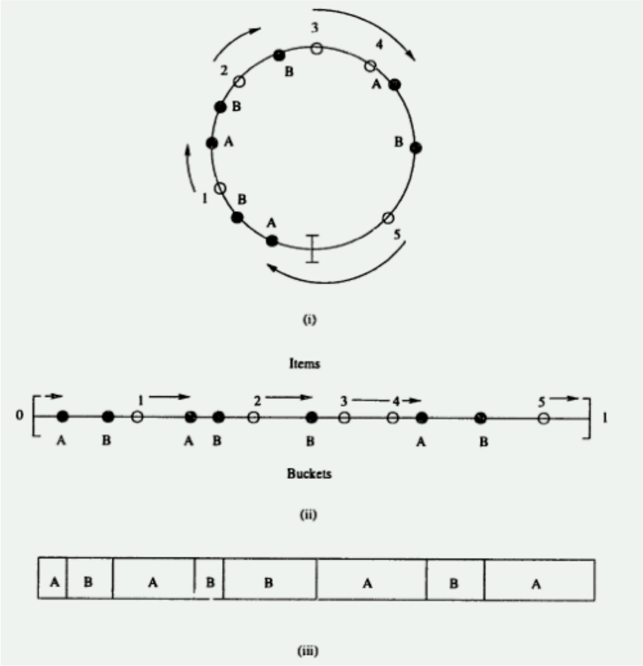
e. QUESTION: All these have extra latency to propagate communication because client does not know where the data is (SEARCH).
    i. How do you get rid of linear (broadcast) or logarithmic search (trees)
    ii. ANSWER: Hashing – map name into much smaller set of locations that is small enough to be cached

f. General solution: hashing
    i. Have the client hash the URL to choose a cache
        1. Does location service without an extra hop by pushing it to client
    ii. H(url) = m*url + b mod q
        1. Problem: if number of nodes change (q), everything moves: (from d+1 mod 4 to d+1 mod 5)
        2. Problem: set of caches may change over time, don't want to lose locality
        3. If clients don't learn of change immediately, don't want to lose performance
        4. 

        5. Each change it set of servers is a **view**, would like data to keep locality across views
    iii. Goal:
        1. - balance – objects are assigned to buckets random

        2. - monotonicity – when a bucket is added or removed, the only objects affected are those mapped to the bucket
        3. - load – objects are assigned to the buckets evenly over a set of views
        4. - spread – an object is mapped to a small number of buckets over a spread of views
        5. QUESTION: Are these realistic goals? How else could you solve it?
            a. LARD: maintain a table of all URLs (but then need to update it …)

g. Solution: Consistent hashing
    i. General idea: add a **layer of indirection** into hashing instead of hashing directly to servers
    ii. Don't hash directly onto a bucket, hash on to a range of real numbers with ranges
    iii. 

    iv. Solution part 2: put each server at multiple locations, so things move from many places
    v. 

    vi. Change of view:

vii.

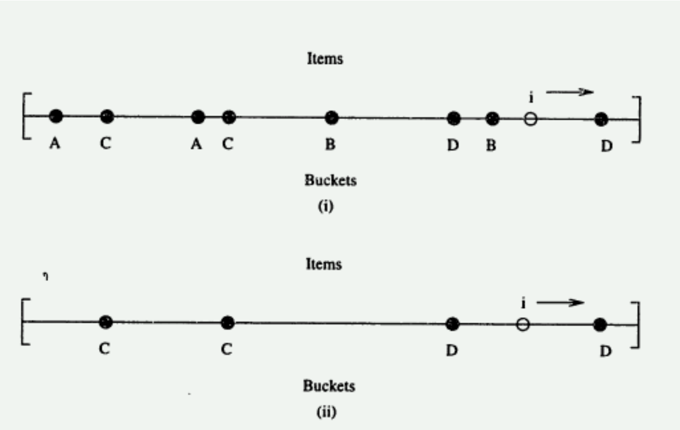


Figure 10: *Monotonicity for the family UC_{random}. In this figure the unit circle is depicted by an interval of length one, which is obtained by cutting the unit circle at an arbitrary point. (i) The mapping of points to the circle for a view $V_2 = A, B, C, D$ (m=2 in this example). The closest bucket point clockwise of i's point is one associated with the bucket D. (ii) For any view $V_1 \in V_2$ containing the bucket D (here $V_1 = C, D$), the point closest to i's point will still be D. .*

h. Details:

i. hash function that takes URLs and outputs a number of 0 ... M

1. URLs and caches are mapped to points on a circle 0 ... M
2. Map caches in multiple places because there are relatively few compared to # of documents, and want even spread.

ii. How to add cache?

1. Move objects that are "closest" on circle to new cache
2. note: map cache on multiple points of circle for uniform
3. distribution of URLs to caches
4. result: each URL is in a small number of caches

iii. How to do lookup?

1. First cache that succeeds hash(U) has document
2. Can store tree with whole range, or partition range and have a tree per partition

iv. How implement?

1. Use DNS: have client hash URL into large number of virtual caches (e.g. 1000), then ask DNS for the physical cache associated with virtual cache
2. DNS contains closest node for each 1000 virtual cache

v. QUESTION: How handle heterogeneous cache hardware?

1. ANSWER: use variable numbers of virtual caches.

i. QUESTION: how compare with LARD?

i. Uses hash to spread load rather than lookup table

        ii. Is not load aware
            1. In their setting, detailed load information not available
        iii. Is scalable: can have multiple DNS servers doing hashing (or all clients)
    j. QUESTION: How handle load?
        i. Cannot do fully distributed (at all clients), as they don't have load information
        ii. Solution: spread hot content across more caches
            1. Step 0: identify hot virtual names in DNS resolver
            2. Step 1: take all names in one of the buckets for a cache and let them go to any server (round robin) (there may be many)
            3. Step 2: reduce number of servers for the virtual name until load goes up
                a. If picked wrong virtual server, will never go up
            4. Step 3: try with another virtual server (bucket)
    k. QUESTION: Is a hash function needed for objects and servers?
        i. Answer: population of objects is large, so want a hash function
        ii. Population of servers is small, so can assign buckets to servers manually or semi-automatically
            1. drop buckets or add buckets based on load…
    l. Question: how handle geography of clients?
        i. Use it when determining virtual caches – can choose ones with a nearby resolver, which returns nearby caches. Done in client script
    m. Fault tolerance:
        i. When a cache fails: can have fixed retry rule (e.g. next node on ring)
        ii. When a DNS server fails: can replicate, clients already know how to contact another DNS server
            1. DNS servers may need to communicate for the hot-page solution
    n. Implementation: Akamai (simplified without geographic location):
        i. Content producer runs a tool to name a document with a hash function
            1. E.g. a604.akamai.com (604 is the hash bucket)
            2. Set time to live to be short (a few seconds) so can respond to load bursts
            3. NOTE: in paper, client did the hash, so was transparent to servers. Here, content producer does hash, so clients not modified.
        ii. DNS lookup of a604 returns result of hash function: set of servers that may contain document
        iii. Note: hashing may not be random, e.g. may try to cluster all objects in an web page to one cache to minimize DNS lookups
5. Comments:
    a. Consistent hashing allows you to use hashing with locality as set of views change
    b. Avoids need to remember everything in front end to keep locality
    c. Lack of state makes dealing with hot spots hard
        i. Hashing spreads most load fairly evenly
        ii. Feedback + increase spread of a virtual server (number of IPs hosting that

virtual server) helps

d. Compare to DHTs:
   i. DHTs typically forward requests multiple times, at minimum once, adding latency
   ii. DHTs don’t necessarily preserve locality, unless they rely on similar techniques, or scale well

e. How work with local cache?
   i. Claims its better to go right to this cache than consult local cache first (local cache adds latency)
   ii. Lacks complete load balancing (true)

f. Adds work to DNS server?
   i. But can add more dns servers …

g. Real world impact:
   i. Idea used in Dynamo for assigning key/values to servers
   ii. Idea used by Akamai for finding caches.