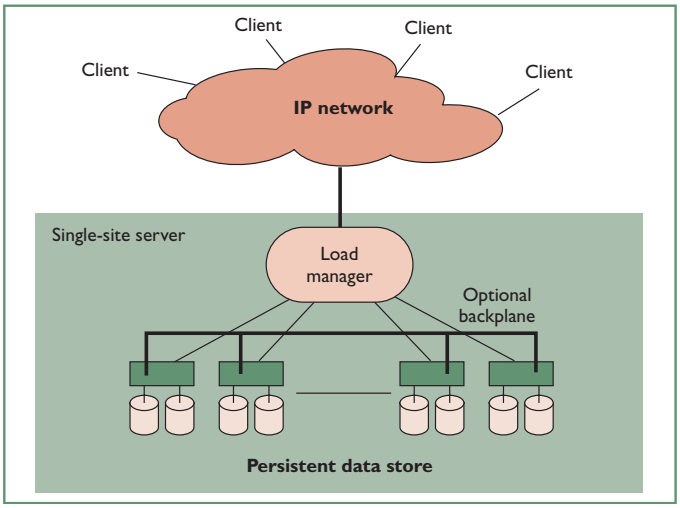
Lecture 4: Large scale

1. Load balancing notes

a. Question after class last time on LARD load balancing algorithm:
    i. When load is about Thigh, move to a node with load below Tlow
    ii. If load is about 2*Thigh, move to any node below Thigh
    iii. Why?
        1. Set 2*Thigh to be maximum number of connections where latency starts to increase
        2. If anybody gets close, start filling low-load ones
        3. If limit # of connections to (n-1)*Thigh + Tlow-1
            a. Then n-1 can be at T high and at least one is below Tlow
            b. Have enough that all are above Tlow
            c. Can have some imbalance (Thigh-Tlow) to prevent too much movement around
        4. Overall:
            a. Want to limit amount of movement, so want a lot of wiggle room
                i. If all nodes have some load, then nothing happens until get to 2*Thigh
            b. Want to avoid lightly-loaded nodes (e.g. new nodes added to system)
                i. Rapidly move things to Tlow
        5. QUESTION: Why choice of max as 2*Thigh?
            a. Somewhat arbitrary, but has subtle feedback role in how often things move.
            b. If set Thigh to Tmax*0.75, then will be slower to shift load to lightly loaded nodes
            c. If set Thigh to Tmax *0.25, then can have a bigger spread (Tlow to 4*Thigh)

2. Giant scale services

a. Questions from reviews
b. Background:
    i. Eric Brewer and some grad students founded inktomi as a search engine using a google-style architecture: commodity workstations and networks (myrinet cluster)
    ii. We read his papers because he writes about his experiences (few others do) and writes for our community
c. What problems addressed in this paper?
    i. Basic architecture
        1. Load-balancing front end
        2. Back-end persistent data store (may be bigger boxes)



3. Best-effort service
4. Where not appropriate?
    a. E-commerce: want to store orders, credit card transactions
5. Why clusters?
    a. Only way to scale to the whole planet
    b. Cheap to buy
    c. Incrementally scalable
    d. Independent failures of small components
6. Cluster architecture:
    a. Use "symmetric design" – really means homogeneous

ii. Load management: LARD & consistent hashing type approaches
1. Layer 4 switches based on TCP ports
2. Layer 7 based on URLs
3. Software switches that persist clients to the same server for session state
4. Client-side failover (e.g. alternate DNS names, alternate cell towers)

d. Availability
i. Metrics
1. MTTF/MTBF = time between failures
2. MTTR = time to repair
    a. Restart app after app crash
    b. Reboot after system crash
    c. Repair /replace hardware after hardware crash
    d. Move workload to another machine
3. Availability/uptime = (MTBF-MTTR)/MTBF = fraction of time you are available to serve data
    a. In a setting with multiple data centers and independent failures, what does this mean?
        i. What a single user sees?
            1. If the internet goes down on their side, they

see zero

ii. Aggregate: of all requsts/ what fraction served?

4. Yield = # queries completed / # queries offered
    a. Aggregate availability
    b. QUESTION: How define for google docs or gmail?
5. Harvest = data available (how much data used for query) / complete data
    a. QUESTION: What is the big idea?
        i. Can degrade service under load.
            1. query fewer sources
            2. Fewer recommendations
            3. Disable features
            4. Example: CNN during 9/11 – reverted to Static, mostly text + images, page
    b. Q: how use in email?
        i. What fraction of inbox/total messages available?
    c. Q: how use in ecommerce?
        i. Reduce number of suggestions
    d. Q: how use in ebay?
        i. Simplified rendering of pages, fewer suggestions or data per page
    e. Q: how use in new york times online?
        i. Simplified pages, less dynamic content

e. Architectures for availability:
    i. Replication: store multiple copies of data
        1. Q: what happens on failure?
            a. Yield goes down – fewer servers to answer results, load is higher on them, cannot serve as many requests.
            b. Harvest stays same (all data still available)
    ii. Partition: split data into smaller chunks
        1. Q: what happens on failure?
            a. Harvest goes down – cannot see all data
            b. Yield stays same (copies of other data stay same), load on them is the same
    iii. QUESTION: What does consistent hashing /LARD do?
        1. Mostly partitioning, replication only for super-hot data
    iv. NOTE: everybody does both
    v. Replication and read/write data
        1. For read-only data, replication adds scalability – can serve more than possible on a single machine
        2. For read/write data, write throughput limited to what a single machine can handle
            a. Must write to all machines, so replication does not

improve throughput
    b. Must partition to the point where load can be handled by a single machine

f. Scalability
    i. DQ principle
        1. Data per query X queries per second = constant for a given cluster/architecture
            a. This is the amount of data you need to process per second, driven by number of machines, disk throughput, network throughput, memory capacity (for caching)
        2. DQ of a cluster is a capacity metric
            a. DQ of a workload is the demand on the cluster. You hope the DQ of the cluster is higher than the DQ of the demand
    ii. How do replication/partitioning and failures affect DQ?
        1. Replication: increase # of queries per second by having more machines answer each query
            a. Failure leads to fewer queries per second
        2. Partitioning: increase amount of data by having more machines store data
            a. Failure leads to less data per query
        3. Result: a failure in either case reduces aggregate capacity the same way
        4. ***Table 1. Overload due to failures.***

Failures	Lost capacity	Redirected load	Overload factor
1	$\frac{1}{n}$	$\frac{1}{n-1}$	$\frac{n}{n-1}$
k	$\frac{k}{n}$	$\frac{k}{n-k}$	$\frac{n}{n-k}$

        5. What happens to the load? Must send it somewhere else (with replication)
            a. If lose 1/n machines, then each other machine must add 1/(n-1) more capacity (with replication)
                i. 5 machines, 1 crashes -> each machine has ¼ more capacity (divide 1 machine over 4)
            b. Other machines have n/(n-1) load (5/4 in our example)

g. What happens at overload?
    i. Overload can happen when unexpected failures (data center) or unexpected workloads (Slashdot effect)
    ii. What bad thing happens?
        1. Congestion collapse: latencies get so long everybody times out and retries

iii. How can you handle?
    1. Must reduce DQ of the load
        a. Queries per second: admission control
            i. Fail low-priority queries
        b. Data per query: incomplete answers
            i. Fewer email messages displayed (in email)
            ii. Fewer tail search results
            iii. Fail complex queries early (lower average data per query)
            iv. Stale data (more caching)

h. Online evolution
    i. Cannot take down an internet service (although AOL used to go down for a few hours every week
    ii. Key question: can versions co-exist?
    iii. Solutions:
        1. Fast reboot: reboot all machines at the same time during off peak hours
            a. Avoid incompatibilities
        2. Rolling upgrade: upgrade in waves, take down 1/#waves at a time
            a. Longer latency, lower impact
            b. Need to support co-existence of versions
            c. *** Most commonly used system
        3. Big flip
            a. Do half the machines at a time, switch from old to new with network switch
    iv. Must support lowered throughput during upgrade, or do during off-peak hours

3. Google

a. QUESTION: What is the goal of the Google search architecture?
    i. High available, scalable, low latency web search
b. QUESTION: What is the envisioned environment?
    i. Multiple data centers
c. QUESTION: What problem does this paper solve?
    i. How to provide **cost efficient** scalable services
d. What is the solution:
    i. Use commodity PCs
        1. QUESTION: Why?
            a. Workload: easily parallelizable, independent so not need high-speed shared memory
            b. Can answer questions in plenty of time, so per-node latency not so important
        2. Provide fault tolerance in software (free once written) rather than expensive hardware
        3. WHY? Needed anyway, might as well leverage.
        4. Stories:

        a. Moved through 5-6 generations of design, driven by cheap per-unit costs
        b. E.g. cork boards: PCBs with cork insulation on metal rails, disks sit on top, 4 computer per board, no parity or ECC
        c. Result: sorting a terabyte of data always had memory corruption, came up with different results every time
    ii. Replication
        1. Multiple machines provide each service, use load balancing to select each one
        2. Allows load on a single piece of data to exceed a single machine capacity
        3. Provides fault tolerance
        4. Make data read-only, so no consistency problems during update – divert queries away from a replica until update in bulk completes
    iii. Partitioning:
        1. Split (***shard***) data across multiple machines: index shards are a portion of the index
        2. Allows a machine to serve less than complete data set
        3. Allows parallel lookup of different parts of index to reduce latency
    iv. Task specialization
        1. Google Web Server: front end to coordinate response
            a. Knows where document and index shards are.
            b. Otherwise stateless – just knows about inflight requests, can learn about where things are after reboot
        2. Index Server: maintains index in memory, looks up matching documents
        3. Document server: stores copy of web documents, returns title, URL, document summary
            a. May store documents on lots of disks, as documents large
    v. Multiple levels of load balancing:
        1. DNS selects a data center
        2. Hardware balancer selects a Google Web server within a cluster
        3. Load balancer choose shard index server
    vi. Homogeneous clusters
        1. Reduces management costs
    vii. What are costs that increase with having lots of machines?
        1. Power: inefficiencies of having lots of fixed cost power
        2. System management: failures scale with # of machines, still need to repair
            a. But with enough, don't need to repair immediately
        3. Balance: need to carefully balance all resources so no general bottleneck
e. Result: spread index through memory of thousands of machines (no going to disk)
    i. Use techniques to compress data in memory (e.g. fewer bits for shorter numbers)
    ii. Sensitive to latency delays
        1. If ask 1000 machines, have to wait for last machine to respond
        2. Random CRON jobs can make pretty much every request slow as a result
    iii. Sensitive to queries of death
        1. Some queries trigger bugs predictable on all servers – thousands of machines die

2. Solution: canaries – send query to one machine first, if succeeds, send to thousands more