Internet Services and Search Engines

Amin Vahdat
CSE 123b
May 2, 2006
Announcements

- Midterm: May 9
- Second assignment due May 15
Lessons from Giant-Scale Services
Service Partitioning

Diagram:
- IP network
- Single-site server
- Load manager
- Myrinet backplane
- Partitioned data store
- Programs connected to the IP network
Case Study: Search

- Map keywords to set of documents containing all words
  - Optionally rank the document set in decreasing relevance
    E.g., PageRank from Google

- Need a web crawler to build *inverted index*
  - Data structure that maps keywords to list of all documents that contains that word

- Multi-word search
  - Perform *join* operation across individual inverted indices

- Where to store individual inverted indices?
  - Too much storage to place all on each machine (esp if you also need to have portions of the document avail as well)
Case Study: Search

- Vertical partitioning
  - Split inverted index across multiple nodes (each node contains as much of index as possible for a particular keyword)
  - Replicate inverted indices across multiple nodes
  - OK if certain portion of document database not reflected in a particular query result (even expected)

- Horizontal partitioning
  - Each node contains portion of inverted index for *all* keywords (or large fraction)
  - Have to visit every node in system to perform full join
Replication versus Partitioning

- **Replication**
  - Any replica can serve any request
  - No need for coherence traffic
  - Failure reduces system capacity but not data availability

- **Partitioning**
  - Nodes are no longer identical so certain requests need to be sent to individual nodes
  - Failure reduces data availability and may reduce capacity
Availability Metrics

- Mean time between failures (MTBF)
- Mean time to repair (MTTR)
- Availability = (MTBF – MTTR)/MTBF
- Can improve availability by increasing MTBF or by reducing MTTR
  - Ideally, systems never fail but much easier to test reduction in MTTR than improvement in MTBF
Harvest and Yield

- **yield** = \( \text{queries completed} / \text{queries offered} \)
  - In some sense more interesting than availability because it focuses on client perceptions rather than server perceptions
  - If a service fails when no one was accessing it…

- **harvest** = \( \text{data available} / \text{complete data} \)
  - How much of the database is reflected in each query?

Should faults affect yield, harvest or both?
DQ Principle

- Data per query * queries per second \(\rightarrow\) constant
- At high levels of utilization, can increase queries per second by reducing the amount of input for each response
- Adding nodes or software optimizations changes the constant
Graceful Degradation

- Peak to average ratio of load for giant-scale systems varies from 1.6:1 to 6:1
- Single-event bursts can mean 1 to 3 orders of magnitude increase in load
- Power failures and natural disasters are not independent, severely reducing capacity
- Under heavy load can limit capacity (queries/sec) to maintain harvest or sacrifice harvest to improve capacity
Graceful Degradation

- Cost-based admission control
  - Search engine denies expensive query (in terms of D)
  - Rejecting one expensive query may allow multiple cheaper ones to complete

- Priority-based admission control
  - Stock trade requests given different priority relative to, e.g., stock quotes

- Reduced data freshness
  - Reduce required data movement under load by allowing certain data to become out of date (again stock quotes or perhaps book inventory)
Online Evolution and Growth

- Internet services undergo rapid development with the frequent release of new products and features.
- Rapid release means that software released in unstable state with known bugs.
  - Goal: acceptable MTBF, low MTTR, no cascading failures.
- Beneficial to have *staging* area such that both new and old system can coexist on a node simultaneously.
  - Otherwise, will have to transfer new software after taking down old software → increased MTTR.
  - Also makes it easier to switch back to old version in case of trouble.
Online Evolution and Growth

- **Fast reboot**
  - Simultaneously “reboot” all machines to new version
  - Simple, but guaranteed downtime

- **Rolling upgrade**
  - Upgrade one node at a time in “wave” moving across cluster
  - Old and new versions must be compatible because they will coexist (hard in practice)

- **Big flip**
  - Update one half at a time
  - Remove one half of system from view of load balancing switch
  - Wait for existing connections to complete
  - Upgrade this half with new software
  - Atomically flip load balancing switch to upgraded software