Overview

- Web caches

- Content distribution networks

Web Caching

- Why cache HTTP objects?
  - Reduce client response time
  - Reduce network bandwidth usage
    - Wide area vs. local area use
  - These two objectives are often in conflict
    - May do exhaustive local search to avoid using wide area bandwidth
    - Prefetching uses extra bandwidth to reduce client response time

Web Proxies

- Also used for security
  - Proxy is only host that can access Internet
  - Administrators makes sure that it is secure
- Performance
  - How many clients can a single proxy handle?
- Caching
  - Provides a centralized coordination point to share information across clients
- How to index
  - Early caches used file system to find file
  - Metadata now kept in memory on most caches

Caching Proxies - Sources for misses

- Capacity
  - How large a cache is necessary or equivalent to infinite
    - On disk vs. in memory → typically on disk
- Compulsory
  - First time access to document
  - Non-cacheable documents
    - CGI-scripts
    - Personalized documents (cookies, etc)
    - Encrypted data (SSL)
- Consistency
  - Document has been updated/expired before reuse
  - Conflict → no such issue
Cache Hierarchies

- Use hierarchy to scale a proxy to more than limited population
  - Why?
    - Larger population = higher hit rate
    - Larger effective cache size
- Why is population for single proxy limited?
  - Performance, administration, policy, etc.
- NLANR cache hierarchy
  - Most popular
  - 9 top level caches
  - Internet Cache Protocol based (ICP)
  - Squid/Harvest proxy

ICP

- Simple protocol to query another cache for content
- Uses UDP – why?
- ICP message contents
  - Type – query, hit, hit_obj, miss
  - Other – identifier, URL, version, sender address (is this needed?)
  - Special message types used with UDP echo port
    - Used to probe server or “dumb cache”
  - Transfers between caches still done using HTTP

Squid Cache ICP Use

- Upon query that is not in cache
  - Sends ICP_Query to each peer (or ICP_Decho to echo port of peer caches that do not speak ICP)
  - May also send ICP_Secho to origin server's echo port
  - Sets time to short period (default 2 sec)
- Peer caches process queries and return either ICP_Hit or ICP_Miss
- Proxy begins transfer upon reception of ICP_Hit, ICP_Decho or ICP_Secho
- Upon timer expiration, proxy request object from closest (RTT) parent proxy
  - Would be better to direct to parent that is towards origin server

Squid

- Simple protocol to query another cache for content
- Uses UDP – why?
- ICP message contents
  - Type – query, hit, hit_obj, miss
  - Other – identifier, URL, version, sender address (is this needed?)
  - Special message types used with UDP echo port
    - Used to probe server or “dumb cache”
  - Transfers between caches still done using HTTP
ICP vs HTTP

- Why not just use HTTP to query other caches?
- ICP is lightweight – positive and negative
  - Makes it easy to process quickly
  - Caches may process many more ICP requests than HTTP requests
- HTTP has many functions that are not supported by ICP
- ICP does not evolve with HTTP changes
- Adds extra RTT to any proxy-proxy transfer

Optimal Cache Mesh Behavior

- Minimize number of hops through mesh
- Each hop add significant latency
  - ICP hops can cost a 2 sec timeout each!
  - Strict hierarchies cost disk lookup, etc.
- Especially painful for misses
- Share across many users and scale to many caches
  - ICP does not scale to a large number of peers
  - Cache and fetch data close to clients

Hinting

- Have proxies store content as well as metadata about contents of other proxies (hints)
- Minimizes number of hops through mesh
  - Size of hint cache is a concern – size of key vs. size of document
- Having hints can help consistency
  - Makes it possible to push updated documents or invalidations to other caches
- How to keep hints up-to-date?
  - Not critical – incorrect hint results in extra lookups not incorrect behavior
  - Can batch updates to peers
Summary Cache

- Primary innovation – use of compact representation of cache contents
  - Typical cache has 8GB of space and 8KB objects → 1M objects
  - Using 16-byte MD5 → 16MB per peer
  - Solution: Bloom filters
  - Delayed propagation of hints
    - Waits until threshold %age of cached documents are not in summary
    - Perhaps should have looked at %age of false hits?

Bloom Filters

- Proxy contents summarize as a M bit value
- Each page stored contributes k hash values in range \([1..M]\)
- Bits for k hashes set in summary
- Check for page = if all pages k hash bits are set in summary it is likely that proxy has summary
- Tradeoff → false positives
  - Larger M reduces false positives
  - What should M be? 8-16 * number of pages seems to work well
  - What about k? Is related to (M/number of pages) → 4 works for above M

Leases

- Only consistency mechanism in HTTP is for clients to poll server for updates
- Should HTTP also support invalidations?
  - Problem: server would have to keep track of many, many clients who may have document
  - Possible solution: leases
- Leases – server promises to provide invalidates for a particular lease duration
- Server can adapt time/duration of lease as needed
  - To number of clients, frequency of page change, etc.

Problems

- Over 50% of all HTTP objects are uncacheable – why?
- Not easily solvable
  - Dynamic data → stock prices, scores, web cams
  - CGI scripts → results based on passed parameters
- Obvious fixes
  - SSL → encrypted data is not cacheable
  - Most web clients don’t handle mixed pages well → many generic objects transferred with SSL
  - Cookies → results may be based on passed data
  - Hit metering → owner wants to measure # of hits for revenue, etc.
  - What will be the end result?

Proxy implementation problems

- Aborted transfers
  - Many proxies transfer entire document even though client has stopped → eliminates saving of bandwidth
- Making objects cacheable
  - Proxy’s apply heuristics → cookies don’t apply to some objects, guesswork on expiration
  - May not match client behavior/desires
- Client misconfiguration
  - Many clients have either absurdly small caches or no cache
  - How much would hit rate drop if clients did the same things as proxies

Problems – Population Size

- How does population size affect hit rate?
  - Critical to understand usefulness of hierarchy or placement of caches
  - Issues: frequency of access vs. frequency of change (ignore working set size → infinite cache)
  - UW/Msoft measurement → hit rate rises quickly to about 5000 people and very slowly beyond that
  - Proxies/Hierarchies don’t make much sense for populations > 5000
    - Single proxies can easily handle such populations
    - Hierarchies only make sense for policy/administrative reasons
Problems – Common Interests

• Do different communities have different interests?
  • I.e. do CS and English majors access same pages? IBM and Pepsi workers?
  • Has some impact → UW departments have about 5% higher hit rate than randomly chosen UW groups
  • Many common interests remain
• Is this true in general? UW students have more in common than IBM & Pepsi workers
• Some related observations
  • Geographic caching – server traces have shown that there is geographic locality to interest
  • UW & MS hierarchy performance is bad – could be due to size or interests?

Overview

• Web caches

• Content distribution networks

CDN

• Replicate content on many servers

Challenges

• How to replicate content
• Where to replicate content
• How to find replicated content
• How to choose among know replicas
• How to direct clients towards replica
  • DNS, HTTP 304 response, anycast, etc.
• Akamai

Server Selection

• Service is replicated in many places in network
• How do direct clients to a particular server?
  • As part of routing → anycast, cluster load balancing
  • As part of application → HTTP redirect
  • As part of naming → DNS
• Which server?
  • Lowest load → to balance load on servers
  • Best performance → to improve client performance
  • Based on Geography? RTT? Throughput? Load?
  • Any alive node → to provide fault tolerance

Routing Based

• Anycast
  • Give service a single IP address
  • Each node implementing service advertises route to address
  • Packets get routed from client to “closest” service node
  • Closest is defined by routing metrics
  • May not mirror performance/application needs
  • What about the stability of routes?

Routing Based

• Cluster load balancing
  • Router in front of cluster of nodes directs packets to server
  • Must be done on connection by connection basis – why?
  • Forces router to keep per connection state
  • How to choose server
    • Easiest to decide based on arrival of first packet in exchange
    • Primarily based on local load
    • Can be based on later packets (e.g. HTTP Get request) but makes system more complex
Application Based
- HTTP support simple way to indicate that Web page has moved
- Server gets Get request from client
  - Decides which server is best suited for particular client and object
  - Returns HTTP redirect to that server
- Can make informed application specific decision
- May introduce additional overhead → multiple connection setup, name lookups, etc.
- While good solution in general HTTP Redirect has some design flaws – especially with current browsers

Naming Based
- Client does name lookup for service
- Name server chooses appropriate server address
- What information can it base decision on?
  - Server load/location → must be collected
  - Name service client
    - Typically the local name server for client
- Round-robin
  - Randomly choose replica
  - Avoid hot spots
- [Semi-]static metrics
  - Geography
  - Route metrics
  - How well would these work?

Naming Based
- Predicted application performance
  - How to predict?
    - Only have limited info at name resolution
  - Multiple techniques
    - Static metrics to get coarse grain answer
    - Current performance among smaller group
  - How does this affect caching?
    - Typically want low TTL to adapt to load changes
    - What does the first and subsequent lookup do?

How Akamai Works
- Clients fetch html document from primary server
  - E.g. fetch index.html from cnn.com
- URLs for replicated content are replaced in html
  - E.g. `<img src="http://cnn.com/af/x.gif">` replaced with `<img src="http://a73.g.akamaitech.net/7/23/cnn.com/af/x.gif">`
  - Client is forced to resolve aXYZ.g.akamaitech.net hostname

How Akamai Works
- Root server gives NS record for akamai.net
- Akamai.net name server returns NS record for g.akamaitech.net
  - Name server chosen to be in region of client’s name server
  - TTL is large
  - G.akamaitech.net nameserver choses server in region
    - Should try to chose server that has file in cache - How to choose?
    - Uses aXYZ name and consistent hash
    - TTL is small
Consistent Hash

- "view" = subset of all hash buckets that are visible
- Desired features
  - Smoothness – little impact on hash bucket contents when buckets are added/removed
  - Spread – small set of hash buckets that may hold an object regardless of views
  - Load – across all views # of objects assigned to hash bucket is small

Consistent Hash – Example

- Construction
  - Assign each of C hash buckets to Klog(C) random points on unit interval
  - Map object to random position on unit interval
  - Hash of object = closest bucket
- Monotone → addition of bucket does not cause movement between existing buckets
- Spread & Load → small set of buckets that lie near object
- Balance → no bucket is responsible for large portion of unit interval

How Akamai Works

1. cnn.com (content provider)
2. DNS root server
3. Akamai server
4. Akamai high-level DNS server
5. Akamai low-level DNS server
6. Closest Akamai server
7. Get index.html
8. Get foo.jpg
9. End user

Akamai – Subsequent Requests

1. cnn.com (content provider)
2. DNS root server
3. Akamai server
4. Akamai high-level DNS server
5. Akamai low-level DNS server
6. Closest Akamai server
7. Get /cnn.com/foo.jpg
8. End user

Next Lecture: P2P

- Peer-to-peer networks
- Assigned reading
  - [Cla00] Freenet: A Distributed Anonymous Information Storage and Retrieval System
  - [S+01] Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications