Web history

  - describes the idea of a distributed hypertext system.
  - a “memex” that mimics the “web of trails” in our minds.
- 1989: Tim Berners-Lee (CERN) writes internal proposal to develop a distributed hypertext system
  - connects “a web of notes with links”.
  - intended to help CERN physicists in large projects share and manage information

Web history (cont)

- 1992
  - NCSA server released
  - 26 WWW servers worldwide
- 1993
  - Marc Andreessen releases first version of NCSA Mosaic Mosaic version released (1)
  - Web (port 80) traffic
  - Over 200 WWW servers worldwide
- 1994
  - Andreessen and colleagues leave NCSA to form “Mosaic Communications Corp” (Netscape).

Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes
  - Heavy-tailed
    - Pareto distribution for tail
    - Lognormal for body of distribution
- Embedded references
  - Number of embedded objects also pareto
  \[ \Pr(X>x) = \left(\frac{x}{x_m}\right)^{-k} \]
  - This plays havoc with performance
- Solutions?
  - Lots of small objects means & TCP
    - 3-way handshake
    - Lots of slow starts
    - Extra connection state
Web Proxy Caches

- User configures browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client

No Caching Example (1)

**Assumptions**
- Average object size = 100,000 bits
- Avg. request rate from institution’s browser to origin servers = 15/sec
- Delay from institutional router to any origin server and back to router = 2 sec

**Consequences**
- Utilization on LAN = 15%
- Utilization on access link = 100%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + milliseconds

No Caching Example (2)

**Possible solution**
- Increase bandwidth of access link to, say, 10 Mbps
- Often a costly upgrade

**Consequences**
- Utilization on LAN = 15%
- Utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + msecs + msecs

W/Caching Example (3)

**Install cache**
- Suppose hit rate is .4

**Consequence**
- 40% requests will be satisfied almost immediately (say 10 msec)
- 60% requests satisfied by origin server
- Utilization of access link reduced to 60%, resulting in negligible delays
- Weighted average of delays
  = .6*2 sec + .4*10msecs < 1.3 secs
HTTP Caching

- Clients often cache documents
  - Challenge: update of documents
  - If-Modified-Since requests to check
    - HTTP 0.9/1.0 used just date
    - HTTP 1.1 has an opaque “entity tag” (could be a file signature, etc.) as well
- When/how often should the original be checked for changes?
  - Check every time?
  - Check each session? Day? Etc?
  - Use Expires header
    - If no Expires, often use Last-Modified as estimate

Example Cache Check Request

GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
If-Modified-Since: Mon, 29 Jan 2001 17:54:18 GMT
If-None-Match: "7a11f-10ed-3a75ae4a"
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.intel-iris.net
Connection: Keep-Alive

Example Cache Check Response

HTTP/1.1 304 Not Modified
Date: Tue, 27 Mar 2001 03:50:51 GMT
Server: Apache/1.3.14 (Unix) (Red-Hat/Linux) mod_ssl/2.7.1 OpenSSL/0.9.5a DAV/1.0.2 PHP/4.0.1pl2 mod_perl/1.24
Connection: Keep-Alive
Keep-Alive: timeout=15, max=100
ETag: "7a11f-10ed-3a75ae4a"

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?
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 misses

Content Distribution Networks (CDNs)

- The content providers are the CDN customers.

  **Content replication**
  - CDN company installs hundreds of CDN servers throughout Internet
    - Close to users
  - CDN replicates its customers’ content in CDN servers. When provider updates content, CDN updates servers

Content Distribution Networks & Server Selection

- Replicate content on many servers
- Challenges
  - How to replicate content
  - Where to replicate content
  - How to find replicated content
  - How to choose among known replicas
  - How to direct clients towards replica

---

Server Selection

- 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
  - How to direct clients to a particular server?
    - As part of routing ➔ anycast, cluster load balancing
      - Not covered 😞
    - As part of application ➔ HTTP redirect
    - As part of naming ➔ DNS

Application Based

- HTTP supports simple way to indicate that Web page has moved (30X responses)
- Server receives 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, but…
  - HTTP Redirect has some design flaws – especially with current browsers

Naming Based

- Client does name lookup for service
- Name server chooses appropriate server address
  - A-record returned is “best” one for the client
- What information can name server base decision on?
  - Server load/location ➔ must be collected
  - Information in the name lookup request
    - Name service client ➔ typically the local name server for client

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

- How is content replicated?
- Akamai only replicates static content (*)
- Modified name contains original file name
- Akamai server is asked for content
  - First checks local cache
  - If not in cache, requests file from primary server and caches file

* (At least, the version we’re talking about today. Akamai actually lets sites write code that can run on Akamai’s servers, but that’s a pretty different beast)

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 chooses server in region
  - Should try to chose server that has file in cache - How to choose?
  - Uses aXYZ name and hash
  - TTL is small → why?

Akamai – Subsequent Requests
Simple Hashing

- Given document XYZ, we need to choose a server to use
- Suppose we use modulo
- Number servers from 1…n
  - Place document XYZ on server (XYZ mod n)
  - What happens when a servers fails? n \rightarrow n-1
    - Same if different people have different measures of n
    - Why might this be bad?

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 random points on mod $2^n$ circle, where, hash key size = n.
  - Map object to random position on unit interval
  - Hash of object = closest bucket
  - Monotone \rightarrow addition of bucket does not cause movement between existing buckets
  - Spread & Load \rightarrow small set of buckets that lie near object
  - Balance \rightarrow no bucket is responsible for large number of objects

Consistent Hashing

- Main idea:
  - map both keys and nodes to the same (metric) identifier space
  - find a “rule” how to assign keys to nodes

Ring is one option.
Consistent Hashing

- The consistent hash function assigns each node and key an \(m\)-bit identifier using SHA-1 as a base hash function.

- **Node identifier**: SHA-1 hash of IP address.

- **Key identifier**: SHA-1 hash of key.

Identifiers

- \(m\) bit identifier space for both keys and nodes.

- **Key identifier**: SHA-1(key)
  
  Key="LetItBe" \(\rightarrow\) SHA-1 \(\rightarrow\) ID=60

- **Node identifier**: SHA-1(IP address)
  
  IP="198.10.10.1" \(\rightarrow\) SHA-1 \(\rightarrow\) ID=123

- How to map key IDs to node IDs?

Consistent Hashing Example

**Rule**: A key is stored at its successor: node with next higher or equal ID.

- IP="198.10.10.1"
- Key="LetItBe"
- Circular 7-bit ID space
- \(N_{123}\) \(\rightarrow\) \(K_{5}\) \(\rightarrow\) \(N_{32}\) \(\rightarrow\) \(K_{60}\) \(\rightarrow\) \(K_{20}\) \(\rightarrow\) \(K_{101}\) \(\rightarrow\) \(N_{90}\)

Consistent Hashing Properties

- **Load balance**: all nodes receive roughly the same number of keys.

- For \(N\) nodes and \(K\) keys, with high probability
  
  - each node holds at most \((1+\varepsilon)K/N\) keys
  - (provided that \(K\) is large enough compared to \(N\))
**Consistent Hash – Example**

- Construction
  - Assign each of C hash buckets to random points on mod \(2^n\) circle, where, hash key size = \(n\).
  - Map object to random position on unit interval
  - Hash of object = closest bucket
- Monotone \(\rightarrow\) addition of bucket does not cause movement between existing buckets
- Spread & Load \(\rightarrow\) small set of buckets that lie near object
- Balance \(\rightarrow\) no bucket is responsible for large number of objects

**Load Balance**

- Redirector knows all CDN server IDs
- Can track approximate load (or delay)
- To balance load:
  - \(W_i = \text{Hash(URL, S_i)}\) for all \(I\)
  - Sort \(W_i\)
  - From high to low find first server with low enough load
- How should “load” be measured?

**Consistent Hashing not just for CDN**

- Finding a nearby server for an object in a CDN uses centralized knowledge.
- Consistent hashing can also be used in a distributed setting
- P2P systems like BitTorrent, e.g., project 3, need a way of finding files.
- Consistent Hashing to the rescue.

**Chord: Design Goals**

- **Load balance**: Chord acts as a distributed hash function, spreading keys evenly over the nodes.
- **Decentralization**: Chord is fully distributed: no node is more important than any other.
- **Scalability**: The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.
- **Availability**: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that the node responsible for a key can always be found.
Lookups strategies

- Every node knows its successor in the ring
- Requires $O(N)$ lookups

Reducing Lookups: Finger Tables

- Each node knows $m$ other nodes in the ring (it has $m$ fingers)
- Increase distance exponentially
- Finger $i$ points to successor of $n+2^{-i} i=1..m$

Faster Lookups

- Lookups are $O(\log N)$ hops

Summary of Performance Results

- Efficient: $O(\log N)$ messages per lookup
- Scalable: $O(\log N)$ state per node
- Robust: survives massive membership changes
Joining the Ring

• Three step process
  • Initialize all fingers of new node
  • Update fingers of existing nodes
  • Transfer keys from successor to new node

• Two invariants to maintain
  • Each node’s finger table is correctly maintained
  • \( \text{successor}(k) \) is responsible for \( k \) (objects stored in correct place)

Join: Initialize New Node’s Finger Table

• Locate any node \( p \) in the ring
• Ask node \( p \) to lookup fingers of new node

1. Lookup(37,38,40,…,100,164)

Join: Update Fingers of Existing Nodes

• New node calls update function on existing nodes

\( n \) becomes the \( i \)th fingerprint of node \( p \) if \( p \) precedes \( n \) by at least \( 2^{i-1} \) and \( i \)th finger of node \( p \) succeeds \( n \).

Update in \( O(\log^2 N) \) expected messages

Join: Transfer Keys

• Only keys in the range are transferred

Copy keys 21..36 from N40 to N36
Handling Failures

- **Problem:** Failures could cause incorrect lookups
- **Solution:** *Fallback:* keep track of a list of immediate successors

```
N120
N10
N102
N85
N80
```

```
Lookup(85)
```

- Use successor list
  - Each node knows $r$ immediate successors
  - After failure, will know first live successor
  - Correct successors guarantee correct lookups

- Guarantee with some probability
  - Can choose $r$ to make probability of lookup failure arbitrarily small

Joining/Leaving overhead

- When a node joins (or leaves) the network, only an fraction of the keys are moved to a different location.

- For $N$ nodes and $K$ keys, with high probability
  - when node $N+1$ joins or leaves, $O(K/N)$ keys change hands, and only to/from node $N+1$

Summary

- Caching improves web performance
- Caching only at client is only partial solution
- Content Delivery Networks move data closer to user, maintain consistency, balance load
- Consistent Caching maps keys AND buckets into the same space
- Consistent caching can be fully distributed, useful in P2P systems using structured overlays