Today

Scalable content distribution

- Infrastructure
- Peer-to-peer

Observations on scaling techniques

---

**P561: Network Systems**
**Week 8: Content distribution**

Tom Anderson
Ratul Mahajan

TA: Colin Dixon

---

**The simplest case: Single server**

1. Where is cnn.com?
2. 204.132.12.31
3. Get index.html
4. Index.html

Single servers limit scalability.

---

**Solution: Use a cluster of servers**

Content is replicated across servers

Q: How to map users to servers?

---

**Method 1: DNS**

DNS responds with different server IPs

---
Implications of using DNS

Names do not mean the same thing everywhere

Coarse granularity of load-balancing
- Because DNS servers do not typically communicate with content servers
- Hard to account for heterogeneity among content servers or requests

Hard to deal with server failures
Based on a topological assumption that is true often (today) but not always
- End hosts are near resolvers
- Relatively easy to accomplish

Method 2: Load balancer

Load balancer maps incoming connections to different servers

Implications of using load balancers

Can achieve a finer degree of load balancing

Another piece of equipment to worry about
- May hold state critical to the transaction
- Typically replicated for redundancy
- Fully distributed, software solutions are also available (e.g., Windows NLB) but they serve limited topologies

Single location limits performance

Solution: Geo-replication

1. Use DNS to map a user to a nearby data center
   - Anycast is another option (used by DNS)
2. Use a load balancer to map the request to lightly loaded servers inside the data center

In some cases, application-level redirection can also occur
- E.g., based on where the user profile is stored
**Question**

Did anyone change their mind after reading other blog entries?

**Problem**

It can be too expensive to set up multiple data centers across the globe
- Content providers may lack expertise
- Need to provision for peak load
Unanticipated need for scaling (e.g., flash crowds)

Solution: 3rd party Content Distribution Networks (CDNs)
- We’ll talk about Akamai (some slides courtesy Bruce Maggs)

**Akamai**

Goal(?): build a high-performance global CDN that is robust to server and network hotspots

Overview:
- Deploy a network of Web caches
- Users fetch the top-level page (index.html) from the origin server (cnn.com)
- The embedded URLs are Akamaized
  - The page owner retains control over what gets served through Akamai
- Use DNS to select a nearby Web cache
  - Return different server based on client location

**Akamaizing Web pages**

Embedded URLs are Converted to ARIs

```html
<html>
<head>
<title>Welcome to xyz.com!</title>
</head>
<body>
<img src="http://www.xyz.com/logos/logo.gif">
<img src="http://www.xyz.com/jpgs/nabbar1.jpg">
<a href="page2.html">Click here to enter</a>
</body>
</html>
```

**Akamai DNS Resolution**

<table>
<thead>
<tr>
<th>End User</th>
<th>Akamai DNS Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser’s Cache</td>
<td>Local Name Server</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15.15.123.5</td>
<td>15.15.123.5</td>
</tr>
<tr>
<td>20.20.123.55</td>
<td>20.20.123.55</td>
</tr>
</tbody>
</table>

**DNS Time-To-Live**

TTL of DNS responses gets shorter further down the hierarchy

- Root: 1 day
- HLDNS: 30 min.
- LLDNS: 30 sec.
DNS maps

Map creation is based on measurements of:
- Internet congestion
- System loads
- User demands
- Server status

Maps are constantly recalculated:
- Every few minutes for HLDNS
- Every few seconds for LLDNS

Measured Akamai performance (Cambridge)

[The measured performance of content distribution networks, 2000]

Key takeaways

Pretty good overall
- Not optimal but successfully avoids very bad choices
  - This is often enough in many systems; finding the absolute optimal is a lot harder

Performance varies with client location

Aside: Re-using Akamai maps

Can the Akamai maps be used for other purposes?
- By Akamai itself
  - E.g., to load content from origin to edge servers
- By others

Aside: Drafting behind Akamai (1/3)

Goal: avoid any congestion near the source by routing through one of the peers
Aside: Drafting behind Akamai (2/3)

Solution: Route through peers close to replicas suggested by Akamai

Aside: Drafting behind Akamai (3/3)

Trends impacting Web cacheability (and Akamai-like systems)
Dynamic content
Personalization
Security
Interactive features
Content providers want user data

New tools for structuring Web applications
Most content is multimedia

Why Akamai helps even though the top-level page is fetched directly from the origin server?

Peer-to-peer content distribution
When you cannot afford a CDN
  - For free or low-value (or illegal) content

Last week:
  - Napster, Gnutella
  - Do not scale

Today:
  - BitTorrent (some slides courtesy Nikitas Liogkas)
  - CoralCDN (some slides courtesy Mike Freedman)

BitTorrent overview
Keys ideas beyond what we have seen so far:
  - Break a file into pieces so that it can be downloaded in parallel
  - Users interested in a file band together to increase its availability
  - “Fair exchange” incents users to give-n-take rather than just take
**BitTorrent terminology**

Swarm: group of nodes interested in the same file

Tracker: a node that tracks swarm’s membership

Seed: a peer that has the entire file

Leecher: a peer with incomplete file

---

**Joining a torrent**

1. A new leecher joins a torrent by sending a join request to the tracker.
2. The tracker sends the peer list and metadata file to the new leecher.
3. The metadata file contains:
   1. The file size
   2. The piece size
   3. SHA-1 hash of pieces
   4. Tracker’s URL

---

**Downloading data**

- Download pieces in parallel
- Verify them using hashes
- Advertise received pieces to the entire peer list
- Look for the rarest pieces

---

**Uploading data (unchoking)**

- Periodically calculate data-receiving rates
- Upload to (unchoke) the fastest k downloaders
  - Split upload capacity equally
- Optimistic unchoking
  - Periodically select a peer at random and upload to it
  - Continuously look for the fastest partners

---

**Incentives and fairness in BitTorrent**

Embedded in choke/unchoke mechanism
- Tit-for-tat

Not perfect, i.e., several ways to “free-ride”
- Download only from seeds; no need to upload
- Connect to many peers and pick strategically
- Multiple identities

Can do better with some intelligence

Good enough in practice?
- Need some (how much?) altruism for the system to function well?

---

**BitTyrant** [NSDI 2006]

[Graph showing rate of original download time and BitTyrant download time]
CoralCDN

Goals and usage model is similar to Akamai
- Minimize load on origin server
- Modified URLs and DNS redirections

It is p2p but end users are not necessarily peers
- CDN nodes are distinct from end users

Another perspective: It presents a possible (open) way to build an Akamai-like system

CoralCDN overview

Implements an open CDN to which anyone can contribute
CDN only fetches once from origin server

CoralCDN components

CoralCDN components

DNS Redirection
- Return proxy, preferably one near client
- Cooperative
  Web Caching

How to find close proxies and cached content?

DHTs can do that but a straightforward use has significant limitations

How to map users to nearby proxies?
- DNS servers measure paths to clients

How to transfer data from a nearby proxy?
- Clustering and fetch from the closest cluster

How to prevent hotspots?
- Rate-limiting and multi-inserts

Key enabler: DSHT (Coral)

DSHT: Hierarchy

DSHT: Routing

A node has the same Id at each level

Continues only if the key is not found at the closest cluster
DSHT: Preventing hotspots

- Proxies insert themselves in the DSHT after caching content
- So other proxies do not go to the origin server
- Store value once in each level cluster
  - Always storing at closest node causes hotspot

DNS measurement mechanism

Server probes client (2 RTTs)

- Coral
- htpprx
- dnsrv

Return servers within appropriate cluster
- e.g., for resolver RTT = 19 ms, return from cluster < 20 ms
Use network hints to find nearby servers
- i.e., client and server on same subnet
Otherwise, take random walk within cluster

CoralCDN and flash crowds

Local caches begin to handle most requests
Hit to origin web server

End-to-end client latency

Scaling mechanisms encountered

- Caching
- Replication
- Load balancing (distribution)
Why caching works?

Locality of reference
- Temporal
  - If I accessed a resource recently, good chance that I’ll do it again
- Spatial
  - If I accessed a resource, good chance that my neighbor will do it too

Skewed popularity distribution
- Some content more popular than others
- Top 10% of the content gets 90% of the requests

Zipf’s law

Observed to be true for
- Frequency of written words in English texts
- Population of cities
- Income of a company as a function of rank
- Crashes per bug

Helps immensely with coverage in the beginning and hurts after a while

Zipf’s law and the Web (1)

For a given server, page access by rank follows a Zipf-like distribution ($\alpha$ is typically less than 1)

Zipf’s law and the Web (2)

At a given proxy, page access by clients follow a Zipf-like distribution ($\alpha < 1; \sim 0.6-0.8$)

Implications of Zipf’s law

For an infinite sized cache, the hit-ratio for a proxy cache grows in a log-like fashion as a function of the client population and the number of requests seen by the proxy

The hit-ratio of a web cache grows in a log-like fashion as a function of the cache size

The probability that a document will be referenced $k$ requests after it was last referenced is roughly proportional to $1/k$. 
Cacheable hit rates for UW proxies

- Cacheable hit rate – infinite storage, ignore expirations
- Average cacheable hit rate increases from 26% to 41% with (perfect) cooperative caching

Hit rate vs. client population

Small organizations
- Significant increase in hit rate as client population increases
- The reason why cooperative caching is effective for UW

Large organizations
- Marginal increase in hit rate as client population increases

Zipf and p2p content [SOSP2003]

- Zipf: popularity(nth most popular object) ~ 1/n^α
- Kazaa: the most popular objects are 100x less popular than Zipf predicts

Another non-Zipf workload

- Fetch-many vs. fetch-at-most-once
- Web objects change over time
  - www.cnn.com is not always the same page
  - The same object is fetched many times by the same user
- P2p objects do not change
  - "Mambo No. 5" is always the same song
  - The same object is not fetched again by a user
Caching implications

- In the absence of new objects and users
  - fetch-many: hit rate is stable
  - fetch-at-most-once: hit rate degrades over time

New objects help caching hit rate

New objects cause cold misses but they replenish the highly cacheable part of the Zipf curve
Rate needed is proportional to avg. per-user request rate

Cache removal policies

What:
- Least recently used (LRU)
- FIFO
- Based on document size
- Based on frequency of access

When:
- On-demand
- Periodically

Replication and consistency

How do we keep multiple copies of a data store consistent?
- Without copying the entire data upon every update
  - Apply same sequence of updates to each copy, in the same order
  - Example: send updates to master; master copies exact sequence of updates to each replica

Replica consistency

While updates are propagating, which version(s) are visible?
DNS solution: eventual consistency
- changes made to a master server; copied in the background to other replicas
- In meantime can get inconsistent results, depending on which replica you consult
Alternative: strict consistency
- before making a change, notify all replicas to stop serving the data temporarily (and invalidate any copies)
- broadcast new version to each replica
- when everyone is updated, allow servers to resume

Eventual Consistency Example
Sequential Consistency Example

Consistency trade-offs

Eventual vs. strict consistency brings out the trade-off between consistency and availability

Brewer’s conjecture:
- You cannot have all three of
  - Consistency
  - Availability
  - Partition-tolerance

Load balancing and the power of two choices (randomization)

Case 1: What is the best way to implement a fully-distributed load balancer?

Randomization is an attractive option
- If you randomly distribute n tasks to n servers, w.h.p., the worst-case load on a server is \( \log n / \log \log n \)
- But if you randomly poll k servers and pick the least loaded one, w.h.p., the worst-case load on a server is
  \( (\log \log n / \log d) + O(1) \)
  \( 2 \) is much better than 1 and only slightly worse than 3

Load balancing and the power of two choices (stale information)

Case 2: How to best distribute load based on old information?

Picking the least loaded server leads to extremely bad behavior
- E.g., oscillations and hotspots

Better option: Pick servers at random
- Considering two servers at random and picking the less loaded one often performs very well