Experiences with CoralCDN: A Five-Year Operational View

Mihir Kulkarni

COS 518: Advanced Computer Systems

April 15, 2019

Some slides taken from Prof. Freedman's presentation.

An open, co-operative, self-organising CDN

Main goal: Make desired content widely available regardless of **publisher's resources**, by organising and using any **co-operative resources**

Previous solutions

1. Client side proxying
2. Throw money at problem
3. Proprietary CDN.

Introduction

1. "Coralising" involves a simple append to the URL.
2. **Transparent** redirection by network of DNS servers to nearby proxy server.
3. Designed to automatically and scalably handle **sudden spikes** in traffic.
4. TBs of data per day, 10s of millions of HTTP reqs

CoralCDN design

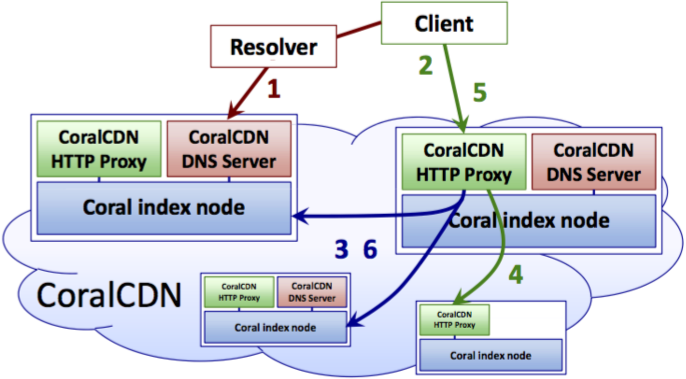


Figure 1: **The steps involved in serving a Coralized URL.**

Usage

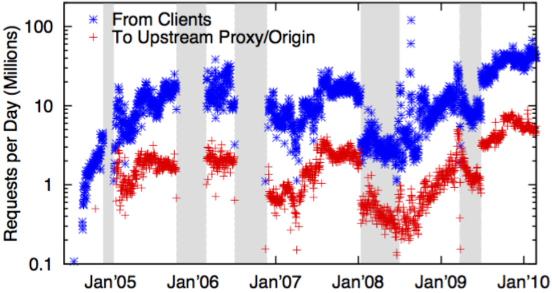


Figure 3: **Total HTTP requests per day during CoralCDN's deployment. Grayed regions correspond to missing or incomplete data.**

Indexing layer

1. Use a ”sloppy” distributed hash table: map key to node, allow scalable lookup in $O(\log n)$ for n node system.
2. Self-reorganising (into locality-sensitive clusters) + failure resistant

Indexing layer

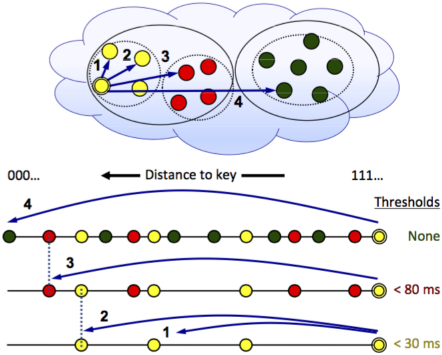


Figure 2: ***Coral's three-level hierarchical overlay structure.*** **A node first queries others in its level-2 cluster (the dotted rings), where pointers reference other caching proxies within the same cluster. If a node finds a mapping in its local cluster (after step 2), its *get* finishes. Otherwise, it continues among its level-1 cluster (the solid rings), and finally, if needed, to any node within the global level-0 system.**

HTTP proxy

Want to minimise load on origin server.

1. Keep local cache, try fetching from neighbouring proxies on miss.
2. Adapting to flash crowds: proxies self-organise into multicast tree for data streaming
3. Cut-through routing (upload portion as soon as download) and optimistic insertion.

Analysing usage

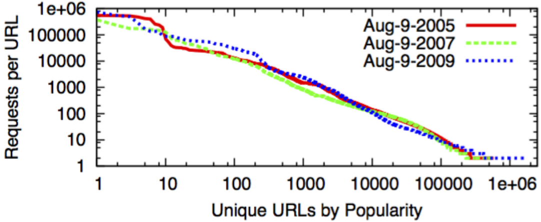


Figure 6: **Total requests per unique URL.**

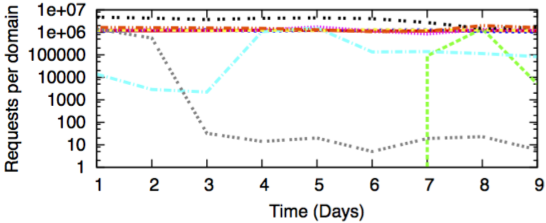


Figure 7: **Requests per top-5 domain over time (Aug 9-18, 2009).**

Use cases

USage scenarios

1. Old content resurrection: Evict unpopular content + proxy kills all cached content in 24h.
2. Heavy tail of unpopular content: Unnecessary: does not reduce load
3. Longterm popular content: Co-operation is overkill: 0.01% URLs account for 49% traffic but need only 14MB.
4. Flash crowds

Handling flash crowds

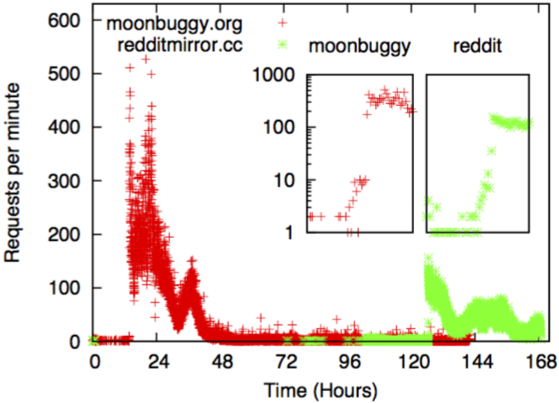


Figure 11: **Mini-flash crowds during August 2009 trace. Each datapoint represents a one-minute duration; embedded subfigures show request rates for the tens of minutes around the onset of flash crowds.**

Handling flash crowds

1. Traffic increases by orders of magnitude in **minutes** not seconds.
2. Large increase rare, % of requests affected not large
3. Can cause several redundant lookups on the origin server: `get` can fail.
4. Mitigated by **optimisitically** inserting node into index, but this hurts performance if the fetch fails.

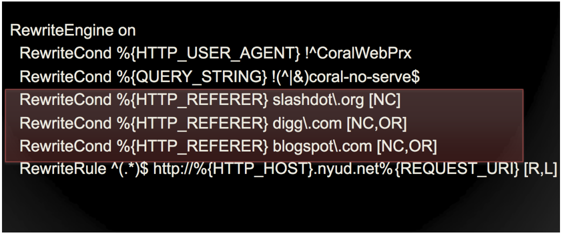
Popular content vs flash crowds

Fundamental tradeoff

1. Most content requested is long-term popular: optimised by little HTTP co-operation + global discovery (DNS)
2. Flash crowds occur on order of minutes: dealt with using regional coop.

Lessons for the Web: API

1. **Interface**: transparency, deep-caching, server-control, ad friendly, forward compatible
2. **Dynamic adoption**: server side redirection via rules written on the origin server



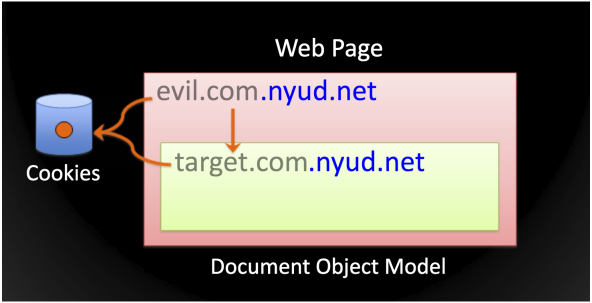
```
RewriteEngine on
  RewriteCond %{HTTP_USER_AGENT} !^CoralWebPrx
  RewriteCond %{QUERY_STRING} !(^|&)coral-no-serve$
  RewriteCond %{HTTP_REFERER} slashdot\.org [NC]
  RewriteCond %{HTTP_REFERER} digg\.com [NC,OR]
  RewriteCond %{HTTP_REFERER} blogspot\.com [NC,OR]
  RewriteRule ^(.*)$ http://%{HTTP_HOST}.nyud.net%{REQUEST_URI} [R,L]
```

Lessons for the Web: Security and resource protection

1. **Limit functionality** restricted class of HTTP requests.
2. **Limit resource use**: Monitor individual clients with sliding window , max file-size.
3. **Blacklist domains+ offload security**

Lessons for the Web: Naming conflation

1. "Same Origin Policy" specifies how scripts from origin affect browser state, and can result in cookies leaking information

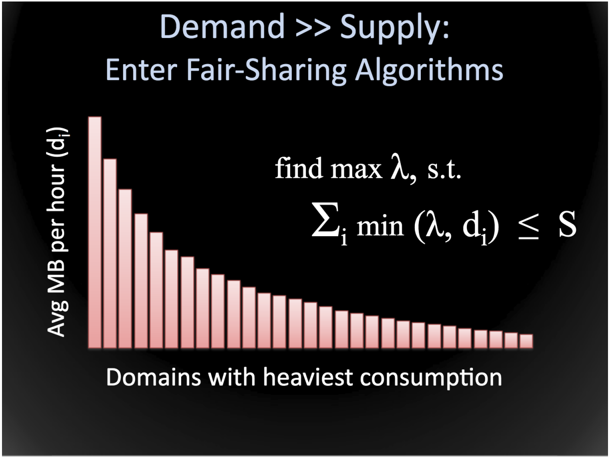


Lessons for CDNs: Design for faulty origins

Accept content conservatively and serve results liberally.

1. Cache **negative** results
2. Serve **stale** content if origin faulty
3. Prevent truncations through whole-file **overwrites**

Lessons for CDNs: Manage oversubscribed bandwidth



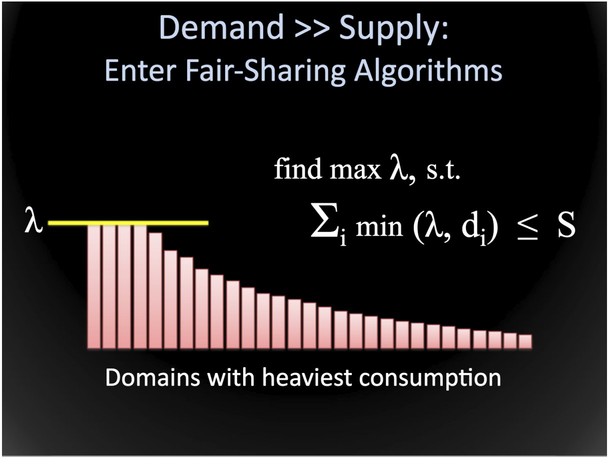
Demand >> Supply:

Enter Fair-Sharing Algorithms

find max λ, s.t.

$$\Sigma_i \min(\lambda, d_i) \leq S$$

Manage oversubscribed bandwidth



Demand >> Supply:

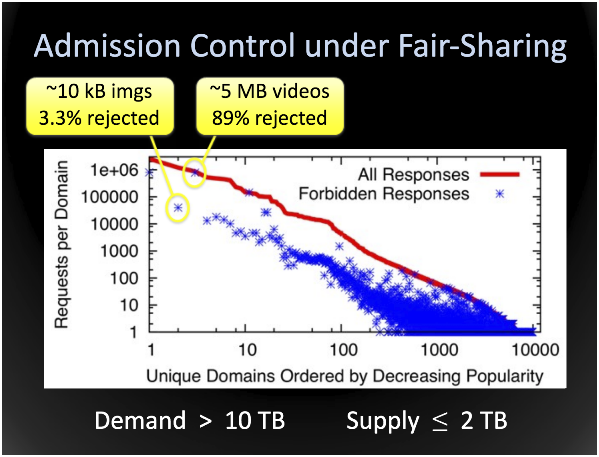
Enter Fair-Sharing Algorithms

find max λ, s.t.

$$\Sigma_i \min(\lambda, d_i) \le S$$

Domains with heaviest consumption

Manage oversubscribed bandwidth



Admission Control under Fair-Sharing

~10 kB imgs
3.3% rejected

~5 MB videos
89% rejected

Demand > 10 TB Supply $\leq$ 2 TB

Conclusion

"Our retrospective on CoralCDN's deployment has a rather mixed message. We view the adoption of CoralCDN as a successful proof-of-concept of how users can and will leverage open APIs for CDN services. But many of its architectural features were over-designed for its current environment and with its current workload: A much simpler design could have sufficed with probably better performance to boot. That said, it is a entirely different question as to whether CoralCDN provides a good basis for designing an Internet- scale cooperative CDN."

Thank you!