Using Latency-Recency Profiles for Data Delivery on the Web

Laura Bright
Louiqa Raschid
University of Maryland
Introduction

• Caching improves data delivery on the web
• Cached data may become stale
• Keeping data fresh adds overhead
  – Latency
  – Bandwidth
  – Server Load
• Existing techniques do not consider client latency and recency preferences
Outline

- Web Technologies
- Existing Solutions
- Latency-Recency Profiles
- Experiments
- Conclusions
Proxy Caches

- Resides between clients and web
- Objects have Time-to-Live (TTL); expired objects validated at server
- Validation adds overhead
- No server cooperation required
Application Servers

- Offload functionality of database-backed web servers
- May perform caching to improve performance
- Servers may propagate updates to cache
Web Portals

- Provide information gathered from multiple data sources
- Problem: updates to objects at sources
  - Update propagation consumes bandwidth
  - Objects at portal may be stale
Consistency Approaches

- Time-to-Live (TTL)
- Always-Use-Cache (AUC)
- Server-Side Invalidation (SSI)
Time-to-Live (TTL)

- Estimated lifetime of cached object
- When TTL expires, cache must validate object at server
- No server cooperation
- Proxy caches
Always Use Cache (AUC)

- Objects served from the cache
- Background prefetching keeps cached objects up to date
- No server cooperation
- Portals, web crawlers
Server Side Invalidation (SSI)

- Servers send updates to cache
- Guarantees freshness
- Increases workload at server
- Application servers
Summary

• Existing techniques do not consider client preferences
• May add unnecessary overhead (latency, bandwidth, or server load)
  – TTL, SSI
• May not meet client recency preferences
  – AUC
• Our goal: consider client preferences and reduce overhead
Outline

• Web Technologies
• Existing Solutions
• Latency-Recency Profiles
• Experiments
• Conclusions
Latency-Recency Profiles

• Used in download decision at cache
• Profile - set of application specific parameters that reflect client latency and recency preferences
• Examples
  – Stock trader tolerates latency of 5 seconds for most recent stock quotes
  – Casual web browser wants low latency; tolerates data with recency two updates
Profile Parameters

- Set by clients
- **Target Latency** \((T_L)\) - acceptable latency of request
- **Target Age** \((T_A)\) - acceptable recency of data
- Examples
  - stock trader: \(T_A = 0\) updates, \(T_L = 5\) seconds
  - casual browser: \(T_A = 2\) updates, \(T_L = 2\) seconds
Profile-Based Downloading

- Parameters appended to requests
- Scoring function determines when to validate object and when to use cached copy
- Scales to multiple clients
- Minimal overhead at cache
Scoring Function Properties

- Tunability
  - Clients control latency-recency tradeoff
- Guarantees
  - upper bounds with respect latency or recency
- Ease of implementation
Example Scoring Function

- **$T$** = target value ($T_L$ or $T_A$)
- **$x$** = actual or estimated value (Age or Latency)
- **$K$** = constant that tunes the rate the score decreases

$$\text{Score} \ (T, \ x, \ K) = \begin{cases} 
1 & \text{if } x \leq T \\
\frac{K}{x - T + K} & \text{otherwise}
\end{cases}$$
Combined Weighted Score

- Used by cache
- \textbf{Age} = estimated age of object
- \textbf{Latency} = estimated latency
- \textbf{w} = relative importance of meeting target latency
- \textbf{(1 - w)} = importance of meeting recency

\[
\text{CombinedScore} = (1 - w) \times \text{Score}(T_A, \text{Age}, K_A) + w \times \text{Score}(T_L, \text{Latency}, K_L)
\]
Profile-Based Downloading

- **CacheScore** - expected score of using the cached object
  \[ \text{CacheScore} = (1 - w) \times \text{Score}(T_A, \text{Age, } K_A) + w \times 1.0 \]

- **DownloadScore** - expected score of downloading a fresh object
  \[ \text{DownloadScore} = (1 - w) \times 1.0 + w \times \text{Score}(T_L, \text{Latency, } K_L) \]

- If **DownloadScore** > **CacheScore**, download fresh object, otherwise use cache
Tuning Profiles

W = 0.5, No firm upper bound

K values control slope
Upper Bounds

$W > 0.5 - \text{Firm Latency Upper Bound}$

$W = 0.6, K_L = 2, K_A = 0.5$

$k_L = 2, k_R = 0.5, w = .6$

W = 0.6, $K_L = 2, K_A = 2$

$k_L = 2, k_R = 2, w = .6$

Download if Latency < 3

Download if Latency < 2
Outline

- Web Technologies
- Existing Solutions
- Latency-Recency Profiles
- Experiments
- Conclusions
Baseline Algorithms

- Time-to-Live (TTL)
  - Estimated lifetime of object
  - Can be estimated by a server or as function of time object last modified
  - Provides most recent data
  - When object’s TTL expires, new object must be downloaded
Baseline Algorithms

• AlwaysUseCache (AUC)
  – minimizes latency
  – If object is in cache, always serve without validation
  – Prefetch cached objects in round robin manner to improve recency
  – Prefetch rates of 60 objects/minute and 300 objects/minute
Baseline Algorithms

- Server-Side Invalidation (SSI)
- SSI-Msg
  - Server sends invalidation messages only
  - Cache must request updated object
- SSI-Obj
  - Server sends updated objects to cache
  - Reduces latency but consumes bandwidth
Trace Data

- Proxy cache trace data obtained from NLANR in January 2002
- 3.7 million requests over 5 days
- 1,365,545 distinct objects, avg size 2.1 KB
- Performed preprocessing
- **Age** estimated using last-modified time
- **Latency** is average over previous requests
- Profiles: $T_L = 1$ second, $T_A = 1$ update
- Cache size range: 1% of world size - infinite
Synthetic Data

- World of 100,000 objects
- Zipf-like popularity distribution
- Update intervals uniformly distributed from 10 min-2 hours
- Workload of 8 requests/sec
- Object sizes 2-12 KB
- Infinite cache
Metrics

- Validations - *messages* between cache and servers
  - *Useful* validations - object was modified
  - *Useless* validations - object not modified
- Downloads - *objects* downloaded from servers
- Stale hits - objects served from cache that were modified at server
## Comparison - Trace Data

<table>
<thead>
<tr>
<th></th>
<th>TTL</th>
<th>AUC-60</th>
<th>AUC-300</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val msgs</td>
<td>252367</td>
<td>378312</td>
<td>1891560</td>
<td>92943</td>
</tr>
<tr>
<td>Useful vals</td>
<td>24898</td>
<td>933</td>
<td>2810</td>
<td>22896</td>
</tr>
<tr>
<td>Useless vals</td>
<td>122074</td>
<td>279349</td>
<td>327776</td>
<td>67601</td>
</tr>
<tr>
<td>Avg. Est. Age</td>
<td>0</td>
<td>18.4</td>
<td>11.1</td>
<td>0.87</td>
</tr>
<tr>
<td>Stale hits</td>
<td>4282</td>
<td>31285</td>
<td>22897</td>
<td>7704</td>
</tr>
</tbody>
</table>
Comparison- Trace Data

Useful Validations

Useless Vals.

Stale Hits
Comparison - Synthetic Data

![Bar chart showing comparisons for validations, downloads, and stale hits for different categories: SSI-Msg, TTL, AUC-300, and Profile.]
Effect of Cache Size

- X-axis - cache size, Y-axis - average latency
- Profile lies between extremes of TTL and AUC
- Profile exploits increased cache size better than TTL
Effect of Cache Size

- X-axis: cache size, Y-axis: number of stale objects
- AUC must prefetch many objects when cache is large
- Profile can scale to large cache size
Effect of Surges

- Surge: client request workload exceeds capacity of server or network
- Two groups of clients:
  - MostRecent: $T_A = 0, T_L = 1$ sec
  - LowLatency: $T_A = 1, T_L = 0$ sec
- 30 second surge period
- Capacity Ratio = available resources / resources required
Effect of Surges

![Graph showing the effect of surges on average latency. The x-axis represents capacity ratio, ranging from 0.2 to 1. The y-axis represents average latency (sec), ranging from 0 to 40. Three lines are plotted: TTL, Profile-MostRecent, and Profile-LowLatency.](image-url)
Related Work

• Refreshing cached data
  – Cho and Garcia-Molina 2000

• WebViews
  – Labrinidis and Roussopoulos, 2000, 2001

• Caching Dynamic Content
  – Candan et al. 2001
  – Luo and Naughton 2001

• Caching Approximate Values
  – Olston and Widom 2001, 2002
Conclusions and Future Work

- Latency-Recency Profiles can reduce overhead while meeting client preferences
- Future work:
  - Implementation
  - Mobile Environments
  - Effects of server cooperation