Exploiting Communities for Enhancing Lookup Performance in Structured P2P Systems

H. M. N. Dilum Bandara and Anura P. Jayasumana
Colorado State University
Anura.Jayasumana@ColoState.edu
Contribution

Community-aware caching scheme to enhance lookup performance in structured P2P systems

1. Build sub-overlays among community members while preserving overlay properties
2. Weighted least frequently used caching based on local statistics

- Enhances both communitywide (23-51%) & system-wide lookup (40%) performance
- Works with structured P2P systems that provide alternative paths to a given destination
- Works with any skewed popularity distribution
- Adaptive to changing popularity
- Need small caches
Motivation

- Many small communities are emerging within P2P systems
- Community – subset of peers that share some similarity
  - Semantic
    - Many BitTorrent communities – music, movies, games, Linux distributions, private communities
  - Geography
    - For 60% of files shared by eDonkey peers, more than 80% of their replicas were located in a single country [Handurukande, 2006]
  - Organizational
    - Peers within an AS, members of a professional organization, group of universities
    - To share resources & limit unrelated external traffic

S. B. Handurukande et al., “Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems,” EuroSys ’06, Apr. 2006.
Motivation (cont.)

• Content popularity in P2P follows Zipf’s-like distribution

• Improve lookup
  – Restructure overlay based on similarity
  – Cache most globally popular content

• However
  1. Communities are not isolated
  2. Individual communities don’t rank high in popularity
  3. Not every node can or interested in caching

## Content Popularity in Communities

1. Communities are not isolated

<table>
<thead>
<tr>
<th>Community*</th>
<th>EX</th>
<th>FE</th>
<th>SP</th>
<th>TB</th>
<th>TS</th>
<th>TE</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>fenopy.com (FE)</td>
<td>0.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seedpeer.com (SP)</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>torrentbit.net (TB)</td>
<td>0.40</td>
<td>0.29</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>torrentscan.com (TS)</td>
<td>0.48</td>
<td>0.33</td>
<td>0.00</td>
<td>0.48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>torrentsection.com (TE)</td>
<td>0.53</td>
<td>0.23</td>
<td>0.00</td>
<td>0.31</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>torrentreactor.net (TR)</td>
<td>0.10</td>
<td>0.08</td>
<td>0.00</td>
<td>0.06</td>
<td>0.09</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>youbittorrent.com (YB)</td>
<td>0.36</td>
<td>0.35</td>
<td>0.00</td>
<td>0.29</td>
<td>0.42</td>
<td>0.20</td>
<td>0.04</td>
</tr>
</tbody>
</table>

EX – extratorrent.com

BitTorrent Communities
2. Communities have different Zipf’s parameters

- \( \alpha = 0.53, 0.66, 0.79, 0.98 \)
- Aggregation of multiple Zipf’s distributions is not necessarily Zipf
- Caching on a structured P2P system with alternative paths [Rao, 2007]

\[
f_r = \frac{1/r^\alpha}{\sum_{n=1}^{N} 1/n^\alpha}
\]

\[
H = \log N - \sum_{r=1}^{C} f_r \log f_r - \log k \cdot L
\]

Structured Overlay – Chord DHT

$n + 2^{i-1}, 1 \leq i \leq m$

$I.\ Stoica\ et\ al.,\ “Chord:\ a\ scalable\ peer-to-peer\ lookup\ service\ for\ internet\ applications,”\ ACM\ SIGCOMM\ ‘01,\ Aug.\ 2001.$
Sub-Overlay Formation

- Goal – not to isolate communities or mix contents
- Each community forms a sub-overlay
  - Form links/fingers to community members
- Enable nodes to identify what’s popular in their community & cache accordingly
  - Forward queries to community members hoping that they may have already cached required contents
Sub-Overlay Formation (cont.)

- Nodes have 1 or more community IDs
  - Communities based on different similarity measures – semantic, geography
  - Support exceptions – user in USA can be a member of a community in India
- Identify community members that are at an exponentially increasing distances in key space
  - Sample nodes pointed by links & their successors
  - Long distant links (large $i$) are more important & easy to find

$B \rightarrow D \rightarrow F = 2$ hops
$B \rightarrow E \rightarrow F = 2$ hops

If E cache F’s content
$B \rightarrow E = 1$ hop

No of distinct node found by probing $i$-th finger & it’s successor
$$2(i + 2 \log N - m) - 1$$
$N$ – No of nodes
$m$ – Key length
$1 \leq i \leq m$
Caching Algorithm

• Cache based on community interest
  – Queries go through community members → Nodes get to know what’s popular in their community

• Local statistics are sufficient to estimate relative popularity
  – Focus on community interest
  – No assumption on popularity distribution

• Weighted least frequently used caching
  – Evaluate demand at arrival of each query \( q \) → Adaptive
  – Weight \( \alpha \) determine bias towards short or long term trends

\[
\begin{align*}
\text{demand}^k_i &= (1 + \alpha) \times \text{demand}^{k-1}_i & \text{If } q \text{ is for } k \\
\text{demand}^k_i &= (1 - \alpha) \times \text{demand}^{k-1}_i & \text{else}
\end{align*}
\]

  – If \( \text{demand}^k > D_{cache} \) → Indicate node’s interest to cache by append to query \( q \)

• Query response is send to query originator & all nodes that want a copy to cache
Caching Algorithm (cont.)

- Reevaluates what keys to cache at arrival of a query
  - Naturally adapts to varying trends of community interests
  - Computationally efficient
- Track contents even if not cached
  - Threshold to remove least popular ones
- $D_{\text{cache}}$ – Caching threshold
  - Prevents cache thrashing
  - $D_{\text{cache}} > \alpha$

```c
void forward(key, msg, nextHop*)
1   If msg.type = PUT //put message
2       return
3   If msg.type = GET //get message
4       addLookup(key) //Track demand
5   If key \in C //In cache
6       sendDirect(msg.source, key, C[key])
7       For each i in msg.cList[] //Send to each cache requester
8           sendDirect(msg.cList[i], key, C[key])
9       nextHop \leftarrow NULL //Drop original get message
10  Else //Not in cache
11     If C.size() = C_{max} //Cache already full
12        key\_lowest \leftarrow \text{getCachedKeyWithLowestDemand}(L[ ])
13        If L[key] > L[key\_lowest] //Higher demand
14           msg.cList[ ] \leftarrow \text{myNodeID} //Request a copy
15           C[key\_lowest]. remove //Remove lowest key
16     Else
17        If L[key] > D_{\text{cache}} //Higher demand
18           msg.cList[ ] \leftarrow \text{myNodeID} //Request a copy
19   For each i in L[ ]
20       If i = key //Increase demand for key
21           L[i] = (1 + \alpha) \times L[i]
22       Else
23           L[i] = (1 - \alpha) \times L[i] //Decrease demand for others
24       If L[i] < D_{\text{remove}} //Very low demand
25           L[i].remove //Remove key
```
Simulation Setup

- OverSim P2P simulation environment
- Sub-overlay formation & caching implemented on top of Chord overlay
- 15,000 nodes
- 10 communities of different sizes
- Different Zipf's parameters
- Queries after system got stabilized – around 2000 sec
- 10 samples
Community, Keys & Query Generation

- Peers know their group ID at initialization
- Each peer
  - Maintain a key index – no capacity limit
  - Maintain a cache – fixed capacity
- Generate fixed set of keys a-priory
  - Peers read keys from a file & store in appropriate nodes
- Queries
  - Use set of Zipf’s parameters observed form BitTorrent

\[ f(r, \alpha, N) = \frac{1}{\sum_{n=1}^{N} \frac{1}{n^\alpha}} \]
Performance Analysis

- Reduced path length
  - Overall system – 40.5%
  - More popular communities – 48-53%
  - Least popular community – 23% reduction (7% with caching)
- Performance depends on skewness
  - $C_1, C_5, & C_6$
- Most queries are responded within few hops

$D_{cache} = 0.12$
$\alpha = 0.1$
$C_{max} = 20$
Performance Analysis (cont.)

- Small cache size per node
- $D_{cache}$ reduce cache thrashing, overhead, & long-term path length
- Rapidly respond to popularity changes
- Better load distribution
  - Max with Chord – 27,574
  - Max with Community Caching – 1,677
Summary

• Community-aware caching solution for structured P2P
  – Allows queries to be forwarded through community members
  – Enable nodes to cache resources that of interest to their community

• Properties
  – Improve both communitywide & system-wide performance
  – Works with any structured P2P system that provides alternative paths to a given destination
    • Preserve overlay bound $O(\log N)$
  – Independent of popularity distribution & how communities are formed
  – Based on local statistics
  – Adaptive
  – Introduces minimum cache storage, network, & computational overhead

• Current/future work
  – Analyze performance under peer churn, heterogeneous caches, & geography based communities
  – In-network community identification & formation
Questions?

Anura.Jayasumana@ColoState.edu
www.cnrl.colostate.edu