Community-Based Caching for Enhanced Lookup Performance in P2P Systems

H. M. N. Dilum Bandara, *Student Member, IEEE* and Anura P. Jayasumana, *Senior Member, IEEE*

Abstract—Large Peer-to-Peer (P2P) systems exhibit the presence of communities based on user interests. Resources commonly shared within individual communities are in general relatively less popular and inconspicuous in the system-wide behavior. Hence, such communities are unable to benefit significantly from caching and replication that focus only on the most dominant queries. A Community-Based Caching (CBC) solution that enhances both communitywide and system-wide lookup performance is proposed. CBC consists of a sub-overlay formation scheme and a Local-Knowledge-based Distributed Caching (LKDC) algorithm. Sub-overlays enable communities to forward queries through their members. While queries are forwarded, LKDC algorithm causes members to identify and cache resources of interests to them, resulting in faster resolution of queries for popular resources within each community. Distributed local caching requires global information (e.g., hop count and popularity of contents) that is difficult and costly to obtain. However, by means of an analysis of globally optimal behavior and structural properties of the overlay, we develop the heuristic-based LKDC algorithm that not only relies on purely local information but also provides close-to-optimal caching performance. CBC is adaptive to changing popularity and user interests, works with any skewed distribution of queries, and introduces minimal modifications and overhead to the overlay network.

Index Terms— Internet applications, network caching, optimization, overlay topology, peer-to-peer

1 Introduction

Modern Peer-to-Peer (P2P) systems share a variety of *resources* such as files, processor cycles, storage capacity, and sensors. Current systems are designed based on either the system-wide behavior, attempting to provide everyone an equal level of service (e.g., average search/download time), or optimized for more dominant users' requirements. In either case, the performance of *lookup* (i.e., the process of searching for resources) degrades as the system sizes continue to grow. Recent studies [1] show that P2P systems in fact consist of many smaller communities. A *community* is a subset of peers that share some similarity in terms of resource semantics, geography, or organizational boundaries. Peers have semantic relationships based on the type of resources they frequently access [1], [2], e.g., many BitTorrent communities dedicated to music, movies, Linux distributions, and games. Users from the same country tend to access similar resources as well [2]. For example, for 60% of the files shared by eDonkey peers, more than 80% of their replicas were located in a single country [2]. Moreover, semantic and geographic similarities are more prominent for moderately popular files. Communities may also arise based on organizational boundaries, e.g., members of a professional organization or a group of universities often forms their own community to share resources and limit unrelated external traffic. Collaborative Adaptive Sensing of the Atmosphere (CASA) [3], an emerging network of weather radars, is one such application where diverse communities of end users (e.g., emergency managers, National Weather Service, media, and scientists) access different subsets of data generated by a distributed set of radars. We can further envision distributed collections of large scientific databases such as Genome sequences, Geographic Information Systems (GIS), weather, census, and economic data that are accessed by communities of users from academic, research, and commercial institutions.

Emerging technological trends such as social networking indicate that we will continue to see the emergence of a large number of small and diverse communities within large P2P systems. Future P2P architectures therefore should support such communities by providing customized services based on their distinct characteristics. Such architectures should allow the emergence, growth, existence, and disappearance of communities on a continual basis, while enabling them to be a part of a global community or a system. Conversely, the P2P system can significantly benefit by taking into account the characteristics and requirements of these communities.

Content popularity profiles in P2P systems follow a Zipf's-like distribution [4], [5], [6]. However, resources popularly shared within an individual community typically do not rank high in popularity in the context of the overall P2P system [2] and often are inconspicuous in the system-wide behavior. Therefore, such communities are unable to benefit from performance enhancements such as caching and replication that focus only on the most popular resources. For example, Beehive [5] and PoP-Cache [6], [7] force a large fraction of peers (in a structured P2P system [8]) to cache the most popular resources regardless of their interests. In spite of requiring large caches and many probing messages to estimate the global popularity, these solutions are inconsiderate of moderately popular resources. Several caching solutions are also available for unstructured P2P systems [9], [10]. However,

• *The authors are with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523. E-mail: dilumb@engr.ColoState.edu and Anura.Jayasumana@ColoState.edu*

due to their random topologies, even the most popular queries are unable to benefit significantly from caching. Instead, several solutions propose to restructure the overlay topology by clustering users/peers with similar interests together [11], [12], [13], [14]. These solutions provide better performance when a user's interests match those of the overall community. However, community membership is not rigid. A user, for example, may belong to multiple communities or switch from a geography-based community to a semantic-based one. Moreover, a community of researchers analyzing the spread of epidemics may access multiple scientific databases such as Genome sequences, GIS, census, and weather data. Our analysis of search clouds from BitTorrent communities confirms that user interests in different communities overlap to some degree. See Section A1 in the supplementary material for a detailed discussion. The prefix 'A' is added when referring to the sections, lemmas, and proofs in the supplementary material (e.g., Section A1, Lemma A1, and Proof A1). Therefore, lookup performance degrades (due to inter-cluster lookup queries) when communities are partially isolated where substantial fraction of queries is for resources outside of a particular cluster. As the communities do not exist in isolation, it is desirable to form one large overlay by combining peers from all the communities such that resources can be efficiently accessed across all the communities. However, existing solutions cannot provide optimum performance under shared communities, and they are not designed to build communities based on incomparable similarity measures such as semantics and geography. Alternatively, mixing of resources from multiple communities is not desirable, as the popularity of individual resources typically subside due to the mixing of many unshared resources (see Section A1). For example, aggregation of two communities with Zipf's-like distributions does not necessarily result in a Zipf's-like distribution unless they have identical set of resources and popularity distributions (Section A1). Therefore, it is important to not only maintain all the communities within a single overlay but also cater to their popularities. Moreover, caching solutions designed for such systems need to be aware of communities' interests, adaptive, and message and storage efficient.

We propose a proactive Community-Based Caching (CBC) solution for structured P2P systems where individual communities form seamlessly and cache resources of interest to them while being in a larger overlay. CBC consists of a sub-overlay formation scheme and a Local-Knowledge-based Distributed Caching (LKDC) algorithm. We first propose a method whereby sub-overlays are formed within the overlay network, enabling communities to forward queries through their members. While the queries are forwarded, LKDC algorithm causes the peers running it to identify and cache resources that are popular within their communities. Therefore, lookup queries for popular resources within a community are resolved faster. Consequently, both the community and system-level lookup performance improve. Distributed Local Caching (DLC) requires global information such as hop count and content popularity that are difficult and costly to obtain. However, by analyzing the globally optimal behavior and taking into account the structural properties of the overlay, we show that it is still possible to develop a close-to-optimal caching solution (namely LKDC) that relies purely on local statistics. CBC is independent of how the communities are formed, adaptive to changing popularity and user interests, and works with any skewed distribution of queries. It is more suitable when users primarily access resources from few communities and the size of a community is moderate to large with respect to the P2P system. Compared to Beehive and PoPCache, which utilize large caches and distributed statistics, CBC caches more distinct resources using smaller caches and utilizes only the local statistics. Simulations based on Chord [15], for example, show 40% reduction in overall average path length with per node cache sizes of 20. Less popular communities reduced the path length by three times compared to system-wide caching.

Problem statement is presented next. Sub-overlay formation and requirements of distributed caching are presented in Section 3. In Section 4, DLC problem, relaxed-DLC problem, and the LKDC algorithm are presented. Simulation setup and performance analysis are presented in Sections 5 and 6, respectively. Section 7 presents concluding remarks. This is an extended version of the paper in [16], and the major extensions include problem formulation, analytical results, and performance analysis.

2 Problem Statement

P2P systems consist of many smaller communities based on semantic, geographic, and organizational interests. However, as discuss in Section A1, sharing among P2P communities (see Table A1) suggests that communities should not be isolated, and conversely combining multiple communities together subsides relative popularities of contents (see Fig. A1). Existing solution space is inadequate as they are limited to either isolating communities or combining all the communities together. Alternatively, better lookup performance can be gained by catering to the popularity of individual communities while being members of a larger P2P system.

Consider a P2P system with a set of $\mathbf{M}$ communities, with community $m \in \mathbf{M}$ consisting of $\mathbf{N}_m$ nodes interested in a set of $\mathbf{K}_m$ resources with normalized popularity f_k, where $k \in \mathbf{K}_m$. Node $n \in \mathbf{N}_m$ has a cache capacity C_n. List of symbols is given in Table 1. Our goal is to find a feasible assignment of cache entries to peers that minimizes the average hop count of each community $m \in \mathbf{M}$.

3 Caching Solution for Communities

We focus on structured P2P systems, as they are appropriate for large-scale implementations due to high scalability and some guarantees on performance [8]. These systems use a Distributed Hash Table (DHT) to index the resources. Each DHT node as well as a resource has its own unique identifier called a *key*. Each resource or its contact information, namely the *value*, is indexed (i.e., stored) at a node having a close by *key* in the key space.

The resources are indexed and retrieved using *put*(*key*, *value*) and *get*(*key*) messages that are forwarded to appropriate nodes using a deterministic overlay. Chord, Kademlia, CAN, and Pastry are some of the well-known solutions that are used to build such an overlay [8]. These solutions typically keep pointers to nodes that are spaced at exponentially increasing gaps in the key space enabling messages to be routed with a bounded path length of $O(\log N)$, where N is the number of nodes in the system.

Let us discuss a specific example using Chord [15], which is considered the most flexible and robust structured P2P system [8]. Chord maps both the nodes and resources into a circular key space (see Fig. 1(a)) using consistent hashing. However, Chord assumes all nodes to be equal partners and does not support any community formation. Based on the *key*, a resource is indexed at its *successor*, i.e., the closest node in the clockwise direction. Each node n maintains a set of pointers, called *fingers*, to nodes that are at $(n + 2^{i-1}) \bmod 2^b$, where $1 \le i \le b$ and b is the key length in bits. For example, node n_E in Fig. 1(a) keeps fingers to nodes n_F, n_G, n_H, and n_J. Routing table at a node consists of these fingers and it is called the *finger table*. Fingers are used to recursively forward a message to a given *key* within a bounded path length of $O(\log N)$. For example, n_E can reach n_L through the route $n_E \rightarrow n_J \rightarrow n_L$. A node may also identify redundant fingers to reduce the latency and enhance robustness, e.g., if n_E knows about n_K, a message may also take the path $n_E \rightarrow n_K \rightarrow n_L$. Nodes can get to know about the demand for different keys by observing the *get()* messages that are forwarded through them. Accordingly, they can either cache the resources corresponding to those keys or their locations.

3.1 Exploiting Community Members to Cache

A *community* is a subset of peers with common interests. Fig. 1(b) illustrates an overlay network having two communities. One of the communities, for example, may be based on semantics while the other may be based on geography. When communities are based on geography or organizational boundaries, nodes can be configured with their unique *Community Identifiers* (CIDs). Therefore, our solution supports communities based on different similarity measures or allows exceptions based on users' interest, e.g., a node in US may connect to a community in India just to access Hindi movies. However, some of the peers may not know their CIDs, may not be aware of the existence of a community with similar interests, or may not even belong to any of the communities. For such cases, solutions such as [11], [17] may be used to assign CIDs to nodes based on their similarity. Dissimilar metrics may be used to group the peers into communities. Only constraint is that each community needs to be identified using a unique CID. For rest of the discussion, we assume such decisions are taken at the application layer [18], outside of the overlay or caching solution. Assuming that each peer knows its community, our goal is to facilitate routing of overlay messages related to a community via its members (by forming a sub-overlay) thus eliminating the inefficiency due to being in a common overlay. During the first couple of hops, the overlay messages tend to hop long distances in the key space and take alternative routes within overlay [7], [8]. Messages converge in the last couple of hops as they approach the destination. Such behavior provides an opportunity for a node to reach its own community members in the first few hops, and then resolve queries using their caches. For example, suppose key k_L indexed at n_L is popular within Community 1 (Fig. 1(b)). n_k is likely to cache k_L as it forwards many queries from its community members to n_L. Consequently, future

TABLE 1
LIST OF SYMBOLS

Symbol	Description
b	Key length in bits
B	Total cache budget
c_k	Cache capacity allocated to key k
C_n	Cache capacity of node n
f_k	Normalized frequency/popularity of key k
$g(c_k)$	Number of hops reduced by caching c_k entries
H_{ave}/h_{ave}	Average hops in a network with/without caching
h_k^n	Number of hops required to resolve a query for key k starting at node n
hop_{max}	Number of hops to forward a community-member-discovery message
K, $\mathbf{K}$	Number/set of keys
m_i	i-th community
N, N_m, $\mathbf{N}$	Number/set of overlay nodes, N_m – in community m
S_k	Size of key k
T_{cache}	Caching threshold
T_{remove}	Remove threshold for entries in lookup table
v_k^n	Value of caching key k at node n
x_k^n	Whether key k is cached at node n. 1 if cached, else 0.
α	Zipf's parameter
β	Parameter use to approximate $g(c_k)$
θ	Weighting factor for query demand
λ_k^n	Demand for key k at node n

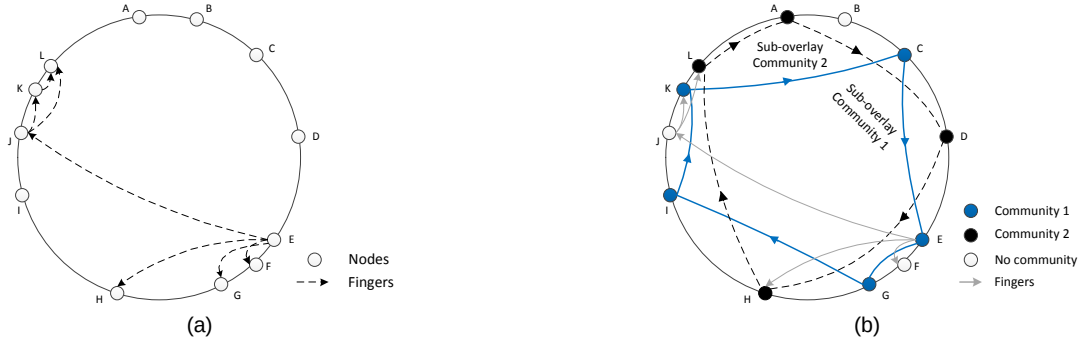

Fig. 1. Chord overlay network: (a) Node connectivity in Chord, (b) Two communities formed on top of the Chord overlay.

queries for k_L can be answered at n_k reducing one-hop. This enables the communities to identify and cache resources of interest to them while enhancing the overall lookup performance. The destination node of a query may or may not belong to the community of the query originator, e.g., n_E and n_L belong to different communities. A querying node n forwards a query through members of its community m under the assumption that "*a resource important to n is also important to other members of m and they may have queried it before n did* [1], [2], [13]. *Therefore, the resource is likely to have been cached in one of the community members along the path*". This assumption and the flexibility of using alternative routes are exploited to design the CBC solution for large-scale P2P systems with multiple communities. First, a mechanism to identify community members is proposed. Then requirements of community-influenced caching are discussed.

3.2 Sub-Overlay Formation

Each node indicates its communities using one or more *CID*s or uses a predefined identifier to indicate that it is not in a community. Based on *CID*s, nodes try to establish stronger connections among community members allowing them to forward queries through sub-overlays. To build a sub-overlay, each node needs to identify other community members that are at approximately exponential distances in the key space. For example, it is useful for n_E to keep pointers to n_G, n_I, and n_K instead of n_F, n_H, and n_J. To take advantage of alternative routes, we need to identify members only for the higher-order pointers/fingers, i.e., ones that point to faraway nodes. To ease the identification process, each node advertises its *CID*s to its successor, predecessor, and other nodes that keep pointers to it. Such advertisements can be piggybacked with overlay maintenance messages. However, given a large number of communities, it is unlikely that a node will identify members using only the advertisements that are sent to specific nodes. Nevertheless, if nodes receiving such advertisements are willing to track those *CID*s within their routing tables, other nodes may query them to find community members. For example, if n_E queries n_J's routing table, it may get to know about n_K. The majority of the structured P2P systems such as Chord, Pastry, and Kademlia maintain many pointers. Therefore, nodes are likely to figure out at least one member for most of the higher-order pointers by sampling a few nodes.

Following modification is proposed to discover community members in Chord. Each node in Chord maintains at least $2 \log_2 N$ fingers [15]. i-th finger ($b - 2 \log_2 N \leq i \leq b$) points to a key space of size 2^{i-1}, i.e., i-th finger can be used to reach any key within a distance of $[2^{i-1}, 2^i)$. It can be shown that both the node pointed to by the i-th finger and its successor can be used to identify $2(i + 2 \log_2 N - b) - 1$ distinct nodes (see Lemma A1). If desired, the finger table of successor's successor may also be probed. Moreover, the probability of finding a community member increases with i, i.e., high-order fingers (see Lemma A2). For example, if the nodes are uniformly distributed in the key space, more community members are likely to be available between pointers to n_H and n_J than between n_F and n_H. Periodic overlay maintenance messages issued by Chord can be used to probe the finger tables for community members. If a member is not found, the message is forwarded to the successor and its finger table is checked. It will be further forwarded to the successor's successor, if a member is still not found. Maintenance message of i-th finger should not be forwarded to the node pointed by the $(i + 1)$-th finger. We limit the number of hops to forward a maintenance message using the parameter hop_{max}. When a member is found, its contact details are piggybacked onto the response to the maintenance message. If the finger table has a limited capacity, nodes may replace original fingers with fingers to community members. Otherwise, both fingers may be maintained for resilience. Once the finger tables are updated, Chord's recursive greedy routing algorithm is used to forward messages to a given key within $O(\log N)$ hops.

If a node changes its community, members of the new community can be identified by refreshing the finger table either immediately or during the next cycle of overlay maintenance messages. Hence, nodes can identify relevant members with minor overhead, and any structured P2P system that provides alternative routers can be used to relay messages through them. Furthermore, worst-case path length bound is still maintained as we preserve the properties of the overlay routing protocol. Our survey of BitTorrent users shows that though users are likely to access contents from multiple communities, 89% of the time they access contents from at most two communities (see Section A3). Therefore, a node needs to maintain fingers only for its primary set of communities hence finger table size is $O(\log N)$. A message may be tagged with the *CID* of the source node so that intermediate nodes can use the suitable set of fingers while forwarding the message.

3.3 Community-Influenced Caching

As the messages are forwarded through sub-overlays, nodes are able to identify and cache resources that are relevant to their communities. Because we focus on communities' interests and preserve the overlay routing properties, local estimation of relative popularities is adequate to decide what a node should cache. For example, consider a node n with $C_n = 1$. If messages mostly come from community members, and k_a is requested more frequently than k_b, n will cache k_a. Sometimes n may observe even more requests for a k_c that is not accessed by its community. This occurs if n is along the path to an overlay neighbor in the Overlay Routing Tree (ORT) that indexes a globally popular key. In such a case, it is useful for n to cache k_c to improve the overall lookup performance. When members of the community interested in k_c (if there is such a community) realize that it is a popular resource, they will add k_c to their own caches. Consequently, n will observe a lower demand for k_c, giving it the opportunity to cache k_a. Therefore, in contrast to previous solutions [5], [6], [7], [9], local statistics are adequate to provide a customized service to each of the communities. Next, we discuss a distributed cache-capacity-allocation mechanism based on the local statistics and structure of the overlay topology.

4. Distributed Caching

We first formulate the Distributed Local Caching (DLC) problem. DLC problem requires global information that is difficult to obtain. Hence, a relaxed version of the problem is formulated based on the overlay properties to answer the two key questions: *where to place cache entries?* and *how many cache entries to create?*. Based on this formulation, a heuristic-based caching algorithm is proposed.

4.1 Distributed Local Caching

In DLC, each overlay node independently decides what keys to cache based on the *get*(*key*) messages that it forwards. For example, suppose n_J in Fig. 1 can cache only one key and each node indexes only one key. If key k_L (indexed at node L) is requested more frequently than k_K, n_J should cache k_L and its corresponding *value*. Therefore, in contrast to previous solutions [5], [6], [7], [9], local statistics are adequate to determine what keys to cache at a node. Query arrivals in P2P systems show flash-crowds, and diurnal and seasonal effects [1], [5]. Therefore, statistics such as periodic, network-wide query counts or arrival rate estimates, used in [5], [6], [7], [9], [19], are inadequate to decide effectively when and what to cache. Moreover, such sampling messages introduce a significant overhead. Instead, nodes can still be made adaptive, if local statistics are collected at different granularities such that long-term and/or short-term popularity changes are properly captured. However, to design an effective solution both the local statistics and overlay topology must be taken into account. For example, suppose n_E forwards five messages to n_F and three messages to n_L through n_J. Based on the local statistics n_E will cache n_F's resources. Therefore, n_E can answer five queries in the future (assuming same query characteristics) and reduce the total hop count by five-hops. However, if n_E caches n_L's resources, it can answer three queries while reducing the total hop count by six-hops. Therefore, it is desirable to cache n_L's resources at n_E, instead of n_F's resources, as the objective of DLC is to improve the lookup performance at a node by reducing the path length of all queries that it forwards. However, reduction in path length cannot be accurately estimated unless topology information is available. Moreover, path length varies when nodes leave and join the network. Such tradeoffs also need to be made when cache capacities of nodes are different and size of resources varies. Hence, overlay topology, cache capacity, size of resources, and their popularity need to be taken into account while determining where to place cache entries and how many cache entries to create.

Consider a P2P system with sets of **N** nodes and **K** keys, and let N and K represent the respective set sizes. Each node is selfish where a node tries to maximize the number of queries that it can answer (irrespective of other nodes) by a caching subset of the (*key*, *value*) pairs. Let $S_k \in Z^+$ be the size of *value* corresponding to *key* $k \in \mathbf{K}$. Let $C_n \in Z^+$ be the cache capacity of node $n \in \mathbf{N}$. At each $n \in \mathbf{N}$, there is a demand $\lambda_k^n \in Z^+$ for $\forall k \in \mathbf{K}$ (e.g., number of queries received over a given period t). Assuming demand for k does not change in the near future, value of caching k depends on its demand and the number of hops that will be reduced due to caching. Therefore, let value $v_k^n = \lambda_k^n h_k^n \in Z^+$ for $\forall k \in \mathbf{K}$, where $h_k^n \in Z^+$ is the number of hops required to resolve a query for k starting at n. Our goal is to minimize the average hop count at a node while satisfying the cache capacity constraint. More formally:

$$\text{minimize} h_{ave}^n = \frac{\text{Totalhops}}{\text{Totalqueries}} = \frac{\sum_{k\in\mathbf{K}} \lambda_k^n h_k^n \left(1 - x_k^n\right)}{\sum_{k\in\mathbf{K}} \lambda_k^n} \qquad (1)$$

$$\text{subject to} \sum_{k\in\mathbf{K}} S_k x_k^n \le C_n$$

where $x_k^n \in \{0,1\}$ determines whether k is cached at node n ($x_k^n = 1$) or not ($x_k^n = 0$). While the optimization problem is NP-complete when the content sizes (S_ks) are different [20], for the purpose of enhancing lookup performance it is sufficient to assume S_ks are small and of similar size. For example, when resources are small (e.g., domain names), a cache entry can be a replica of the resource. When resources are large (e.g., files), then a cache entry can point to the location(s) of the resource. Therefore, for practical purposes, we can assume $S_k = 1$ for $\forall k \in \mathbf{K}$. Eq. (1) can be also interpreted as maximizing the hop count reduction. Then the DLC problem for the purpose of content lookup can be formulated as follows:

$$\text{maximize} R = \sum_{k\in\mathbf{K}} \lambda_k^n h_k^n - \sum_{k\in\mathbf{K}} \lambda_k^n h_k^n \left(1 - x_k^n\right) = \sum_{k\in\mathbf{K}} \lambda_k^n h_k^n x_k^n \qquad (2)$$

$$\text{subject to} \sum_{k\in\mathbf{K}} x_k^n \le C_n$$

This problem can be solved using a greedy algorithm that caches the set of resources with the highest value v_k^n. However, to calculate v_k^n we still need h_k^n which is difficult to obtain. Next, by analyzing the globally optimal behavior and taking into account the structural properties of the ORT, we show that it is possible to develop a close-to-optimal, caching solution without finding h_k^n.

4.2 Global-Knowledge-Based Distributed Caching

We formulate a relaxed version of the DLC problem in (2) that yields an analytical approximation to determine a suitable cache placement strategy. We consider the structure of ORT, as the topology is important in determining *where* and *how much* to cache. Fig. 2(a) illustrates a Chord ring with 32 nodes occupying the entire address space (b = 5, address range [0, 31]). Fig. 2(b) illustrates the asymmetric ORT corresponding to key seven (k_7) for the general case where each node sends one *get*(k_7) message. Branch weights indicate the number of messages forwarded from each node to its parent. A Chord ring with all the nodes is considered to simplify the following discussion. It was also confirmed (through simulations) that such an asymmetric tree exists even when only a small number of nodes are randomly mapped to the ring (i.e., $N \ll 2^b$). Asymmetric ORT explains why the path length in Chord is bounded by $O(\log_2 N)$, average path length is $\frac{1}{2}\log_2 N$, and bell-shaped distribution of path lengths (e.g., see the number of branches at each level of the tree listed on right of Fig. 2(b)). To our knowledge, the relationship between asymmetric ORT and above properties was not observed in prior studies. Similar ORTs can be

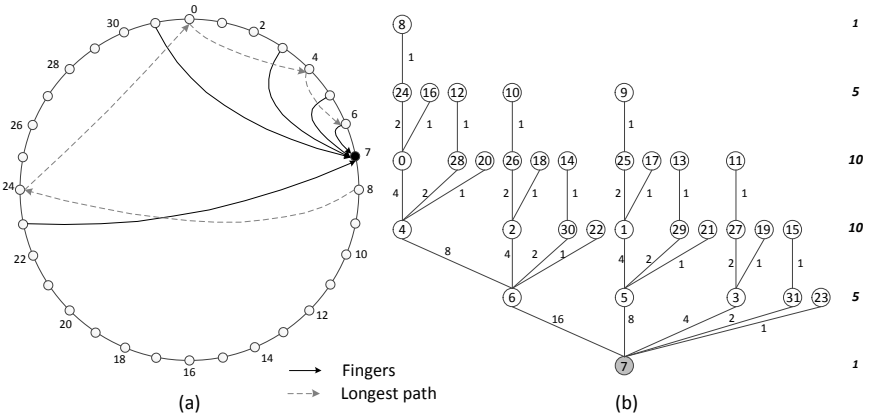

Fig. 2. (a) Chord overlay with 32 nodes. Only the predecessors of node 7 and longest path is shown. (b) Overlay routing tree of node 7.

formulated for other structured P2P systems as well.

Next, we determine the best cache placement strategy given the asymmetric ORT. Node six (n_6) forwards the largest number of messages to n_7. Hence, if there is only one cache entry, it should be placed at n_6 such that all 16 lookup messages can be answered while reducing the number of hops by 16. Suppose there are two cache entries. First entry should be placed at n_6 and the remaining entry can be placed at either n_5 or n_4. If the second cache entry is placed at n_5, it reduces eight-hops and the total reduction is 16 + 8 = 24-hops. Instead, if the second cache entry is placed at n_4, it reduces two-hops for eight messages (between n_4 and n_6) and one-hop for remaining eight messages (between n_6 and n_7). Still the total reduction is 24-hops. If there are three cache entries, they should be placed at n_6, n_5, and n_4, and the total reduction is 32-hops. Similarly, when there is a fourth cache entry, it can be placed at n_3, n_2, n_1, or n_0, and 4-hop will be reduced. Therefore, the number of hops reduced by adding more and more cache entries follow the sequence 16 + 8 + 8 + 4 + 4 + 4 + 4 + 2 + … If there are c_k cache entries allocated to k, it reduces the number of hops by (Lemma A3):

$$g(c_k) = \frac{N}{2}\sum_{i=1}^{c_k} 1/2^{\lfloor \log_2 i \rfloor}, \quad c_k \geq 1 \tag{3}$$

$g(0) = 0$. Average reduction in hop count is given by $g(c_k)/N$. Given that ORT is asymmetric, this is the best cache placement strategy. PoPCache [6], [7] assumed that the ORT is symmetric and cache entries were placed at level 2 nodes only after placing them at all the level 1 nodes. For example, a cache entry was placed at n_4 only after n_6, n_5, n_3, n_{31}, and n_{23}. Therefore, PoPCache did not effectively utilize the ORT to place cache entries.

Given the ORT-based cache placement strategy, we now determine how many cache entries to create for each key k ($c_k \in Z^+$) based on its popularity. Each key has a corresponding ORT and each overlay node belongs to multiple ORTs. Depending on a node's position in different ORTs and popularity of keys, it may have to cache multiple (*key*, *value*) pairs. However, how much a node can cache depends on its cache capacity C_n. We relax the per node cache capacity constraint in (2), such that caching behavior with respect to each key can be examined separately. We still assume a fixed global cache budget B. $B = NC_{ave} \ll NK$, where C_{ave} is the average cache capacity of a node. Furthermore, assume the global popularity of keys is known and the normalized popularity of k is f_k ($0 < f_k \leq 1$). Keys are ordered according to their popularity $f_1 \geq f_2 \geq f_3 \geq \ldots \geq f_K$. Also, assume each resource is of unit size (e.g., address of a node that indexes a resource). We name this scheme as Global-Knowledge-based Distributed Caching (GKDC). Corresponding relaxed GKDC-optimization problem can be formulated as (Lemma A4):

$$\begin{aligned} &\text{minimize} H_{ave} = h_{ave} - \frac{1}{N}\sum_{k\in K} f_k g(c_k) \\ &\text{subject to} \sum_{k\in K} c_k = B, \quad c_k \leq N-1 \quad \forall k \in K \end{aligned} \tag{4}$$

where h_{ave} is the average path length of the overlay network without caching. Summation term in the objective function indicates the number of hops reduced due to caching. First constraint captures the global cache capacity constraint. Second constraint bounds c_k, as it is not useful to cache the same key multiple times at a node. While formulating the optimization problem, PoPCache used the upper bound $O(\log N)$ instead of h_{ave}, did not bound c_k, and assumed the ORT is symmetric. Beehive does not consider a bounded B, structure of the ORT, and support only a Zipf's-like distribution. Optimization problem can be restated as maximizing the hop count reduction:

$$\begin{aligned} &\text{maximize} \sum_{k\in K} f_k g(c_k) \\ &\text{subject to} \sum_{k\in K} c_k = B, \quad c_k \leq N-1 \quad \forall k \in K \end{aligned} \tag{5}$$

Theorem 1. *H_{ave} is minimized when*

$$c_k = \begin{cases} N-1 & \text{if } k \leq l \\ \dfrac{f_k\left(B - l(N-1)\right)}{P(l,K,\alpha)} & \text{otherwise} \end{cases}$$

where $P(l, K, \alpha)$ is the sum of popularity of last $K - l$ keys and l is the smallest key identifier that satisfies

$$\frac{f_{l+1}\left(B - l(N-1)\right)}{P(l,K,\alpha)} < N-1$$

See Section A1 for the proof. Theorem 1 suggests that the most popular l keys should be cached in all the nodes, and the remaining cache capacity $B - l(N-1)$ should be allocated in proportion to the popularity of rest of the keys. This is the best cache capacity allocation strategy

given that the ORT is asymmetric. Therefore, in contrast to PoPCache, we are able to fully utilize the available cache capacity B and provide a tight bound to H_{ave}. Moreover, Theorem is valid for any popularity distribution. H_{ave} can be determined by substituting answer from Theorem in (4). Moreover, GKDC does not require finding hop count information. Correctness of the analytical solution and comparison with PoPCache are presented in Section 6.1. While these bounds are valid for Chord, we believe a similar approach will yield the bounds for other structured overlays by taking into account the structure of their ORTs.

4.3 Local Knowledge-Based Distributed Caching

Cache placement and capacity allocation strategies obtained using the analysis of GKDC can be used to develop a heuristic-based algorithm for the DLC problem. Asymmetric ORT indicates that a key should be cached first at the node that forwards the largest number of messages. Then at one of the nodes that forwards the second largest number of messages, and so on. This will result in a consistent reduction in path length of messages. Theorem 1 says that the cache capacity should be allocated in proportion to the global popularity of keys. In DLC, nodes are not aware of the global popularity of keys. However, most popular keys are evident throughout the ORT while moderately popular ones are evident at lower levels of the ORT. Therefore, a good approximation to the proportional allocation can be obtained using a heuristic that captures the relative popularity of keys at a node. For example, if a node with cache capacity $C_n = 1$ forwards messages of k_a more frequently than messages of k_b, it should cache k_a. Such a heuristic also enables the enforcement of per node capacity constraint where a node will cache locally most popular C_n keys. Furthermore, local statistics can be collected at different granularities such that long-term and/or short-term popularity changes are properly captured. We propose a purely Local Knowledge-based Distributed Caching (LKDC) algorithm based on the Least Frequently Used (LFU) algorithm.

Fig. 3 illustrates the proposed LKDC algorithm described using the common API in [18]. Each node n has a *cache*, which can store up to C_n (*key*, *value*) pairs. An overlay message (*msg*) has a *source* node, message *type* (*put* or *get*), and a *key*. For each *get*(*key*) message that a node receives, we track *key*'s *demand* $\in R^+$ using a (*key*, *demand*) pair which is stored in the lookup table *LT*. *get*() messages also maintain a list of nodes (*cList*) that have decided to cache the resource. *forward()* is an upcall, to the DHT layer, invoked at each node that forwards a message. It enables intermediate nodes to cache, collect statistics, or drop messages. *put*(*key*, *value*) messages are handled as usual. When a *get*(*key*) message is received, each node keeps track of the demand for the corresponding *key* regardless of whether it is already cached or not (line 4 in Fig. 3). The local *cache* is then checked to see whether the *msg* can be answered. If so, replies are directly sent to the *source* node and to the list of nodes in the *cList* that are interested in caching the resource (lines 5-8). *msg* is then dropped (line 9). If the *key* is not in the *cache*, the node tries to determine whether it is useful to get a copy of the resource. If the

```
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 ∈ cache                    //In cache
6         sendDirect(msg.source, key, cache[key])
7         For each i in msg.cList[ ]     //Send to each cache requester
8            sendDirect(msg.cList[i], key, cache[key])
9         nextHop ← NULL                 //Drop original get message
10     Else                              //Not in cache
11        If cache.size( ) = C_n         //Cache already full
12           key_lowest ← getCachedKeyWithLowestDemand(LT[ ])
13           If LT[key] > LT[key_lowest]          //Higher demand
14              msg.cList[ ] ← myNodeID           //Request a copy
15              delete cache[key_lowest]          //Remove lowest
16        Else
17           If LT[key] > T_cache                 // Higher demand
18              msg.cList[ ] ← myNodeID           //Request a copy
void addLookup(key)
19  For each i in LT[ ]
20     If i = key                        //Increase demand for key
21        LT[i] = (1 + θ) × LT[i]
22     Else                              //Decrease demand for others
23        LT[i] = (1 – θ) × LT[i]
24     If LT[i] < T_remove               //Very low demand
25        delete LT[i]                   //Remove key
26  If key not in LT
27     LT[key] ← θ
```

Fig. 3. Local knowledge-based distributed caching algorithm.

cache is already full, it checks whether the given *key* has a higher demand than the LFU cache entry (lines 12-13). A node may also request a copy of the resource, if the *cache* is not fully occupied and the demand is above the caching threshold T_{cache} (lines 17-18). In either case, the node appends (piggyback) its identifier to the *cList* in the *msg*. The *msg* is then forwarded to the next node. Intermediate nodes may also append their identifiers to the *msg*, if they also decide to cache the resource. The threshold T_{cache} reduces the caching overhead and cache thrashing.

If resources are small and relatively static (e.g., domain names), then a cache entry can be a replica of the resource. If resources are large, mutable, or the set of peers having a resource is dynamic (e.g., files and processor cycles), then cache entries can point to sources of the resources. Therefore, our caching scheme can locate all copies of small and relatively stable resources, or point to a subset of large, mutable, or dynamic resources. The caching algorithm works with any distribution of queries and gain better performance when queries are highly skewed.

Using the *addLookup* function, a node also tracks the demand for keys that are not in the cache but forwards messages through it. Thus, LKDC algorithm is a *perfect LFU* algorithm. It enables the network to rapidly adapt to varying popularity and arrival patterns. However, it is important to properly balance the past and new information to reduce the caching overhead. Therefore, whenever a new message arrives, a node multiplies the current demand of the key by a weighting factor $(1 + \theta)$, $0 \le \theta \le 1$. *demand* of all other keys in *LT* is multiplied by $(1 - \theta)$. If a key appears for the first time its demand is set to θ (lines 26-27). When θ is close to zero, the algorithm is biased

towards the past thus effectively responding to long-term trends. When θ is closer to unity, bias is towards the current information, thus a node responds to rapid changes. θ and T_{cache} control the adaptability of the caching solution while minimizing unnecessary cache requests. It is recommend to set $T_{cache} > \theta$ to reduce cache thrashing. Though perfect LFU algorithms are known to take better caching decisions, they have a higher overhead as LT can grow arbitrarily large. A threshold (T_{remove}) is used to remove keys without sufficient *demand* thereby limiting the size of LT. Computational cost of the algorithm is $O(size(LT))$. As LT is not large, we can afford to execute the algorithm every time a *get()* message arrives. Therefore, it can rapidly adapt to changing popularity, message arrival patterns, and piggyback cache requests on *get*() messages while incurring minimum overhead.

5 Simulator

To validate the analysis in Section 4, we first simulated an overlay network with 1,000-5,000 nodes using the OverSim P2P simulator [21]. Caching algorithms were implemented on top of Chord, and the Zipf's parameter α was varied from 0.5 to 1.5. For comparison, PoPCache was also simulated with accurate global popularity of keys. CBC is simulated using a 15,000-node network with ten communities. Size of our network is either comparable or larger than the prior studies such as [5] and [6]. Nodes were assigned to different communities as shown in Table 2. Parameters for each community were selected to observe behavior under different scenarios. Zipf's and similarity parameters were selected based on our own observations in Table A1 and [4], [5]. To simplify the performance analysis, a static network is assumed and queries were issued only after the network was stabilized (around 2,000 s). Each node issued queries based on a Poisson distribution with a mean inter-arrival time of 15 s. Based on the simulations, we observed that it is sufficient to set $T_{remove} = \theta^{10}$ (as query demands are weighted) to gain better performance while limiting the lookup table size to 50-80 entries. Results are based on ten samples, which were sufficient to attain average number of hops within ±5% accuracy and 95% confidence level.

6 Performance Analysis

We first validate the analytical results obtained in Section 4 using a network with a single community. Then performance of community caching is evaluated.

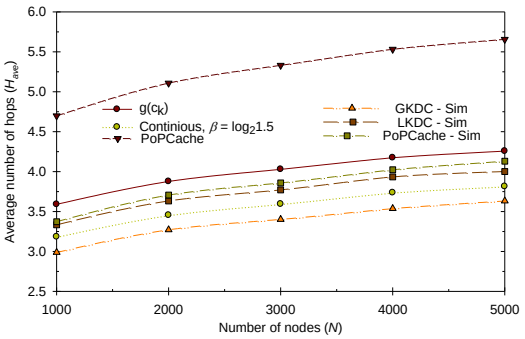

Fig. 4. Average hop count vs. number of nodes. K = 30N, α = 1.0, C_n = 20, T_{cache} = 0.12.

6.1 Local-Knowledge-Based Distributed Caching

Fig. 4 compares the analytical and simulation results while varying N. Our analytical solution using discrete $g(c_k)$ (3) and its continuous approximation using $\beta = \log_2 1.5$ closely match the simulation results. $g(c_k)$-based model provides an upper bound to all the simulation results. Difference between LKDC algorithm (*LKDC-Sim*) and GKDC (*GKDC-Sim*) is ~0.4-hops. Therefore, LKDC provides a desirable caching solution without the cost of estimating global popularity, structure of the ORT, or relaxing the per node cache capacity constraint. The PoPCache analytical model is derived using the upper bound $O(\log N)$ for path length. Therefore, the resulting average hop count (H_{ave}) is very high, e.g., H_{ave} = 8.82 for the setup in Fig. 4. Hence, as a simple correction, we replaced the upper bound with h_{ave}. Still, the corrected PoPCache analytical model overestimates H_{ave} by 1.1-1.4-hops (see Fig. 4). Performance of *PoPCache-Sim* (with accurate global popularity information) and *LKDC-Sim* is similar. However, *PoPCache-Sim* placed 258 keys in one of the nodes whereas *LKDC-Sim* placed only 20 keys in a node (reduced by an order of magnitude). Therefore, our algorithm is more useful as it does not require relaxing the cache capacity constraint or sampling messages to estimate popularity. *GKDC-Sim* has the lowest H_{ave}. It was realized that, though the ORTs in our simulations were also asymmetric, branch weights were somewhat off from Fig. 2(b). For example, most popular path was carrying 55-65% of the queries though our model assumes 50%. Therefore, the most popular branch answers more quires (particularly useful for moderately popular keys) consequently reducing H_{ave}. This explains why the analytical model provides a useful upper bound to simulation results. It was observed that *LKDC-Sim* naturally arranges cache entries

TABLE 2
Configuration of Different Communities.

Community	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
No of nodes (apx.)	600	600	600	1,200	1,200	1,200	1,200	1,200	2,400	4,800
Zipf's parameter	0.85	0.95	1.10	0.5	0.80	0.80	1.0	0.90	0.90	0.75
No of keys	40,000	30,000	30,000	40,000	40,000	40,000	50,000	50,000	50,000	50,000
Similarity with community (x)	0.2 (m_8)	0	0.1 (m_7)	0.2 (m_9)	0.3 (m_8) 0.5 (m_7)	0	0.1 (m_3) 0.5 (m_5)	0.3 (m_5) 0.2 (m_1)	0.4 (m_1) 0.2 (m_4) 0.3 (m_{10})	0.3 (m_9)
Queries for k_1	4,516	8,535	17,100	603	6,454	6,454	21,059	11,956	23,911	17,030

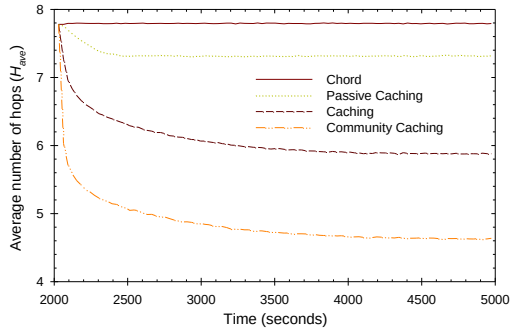

Fig. 5. Average hop count vs. time. Bucket size = 30 *s*, C_n = 20, θ = 0.1, T_{cache} = 0.12, and hop_{max} = 4.

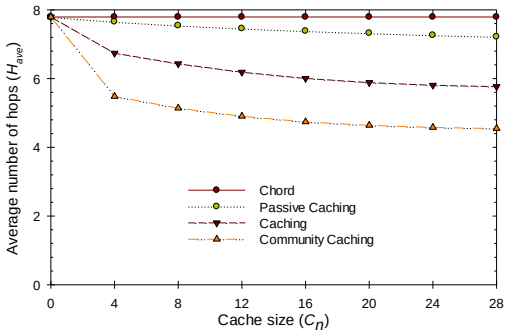

Fig. 7. Impact of cache size (C_n) on average hop count. Steady state results with θ = 0.1, T_{cache} = 0.12, and hop_{max} = 4.

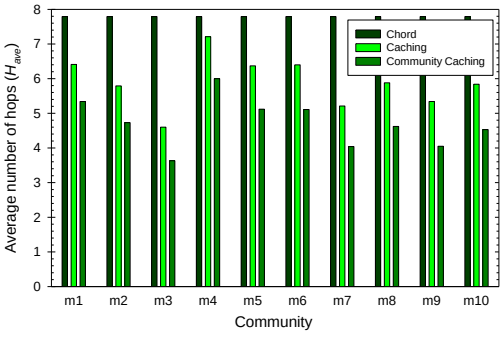

Fig. 6. Average hop count observed by each community at the steady state. C_n = 20, θ = 0.1, T_{cache} = 0.12, and hop_{max} = 4.

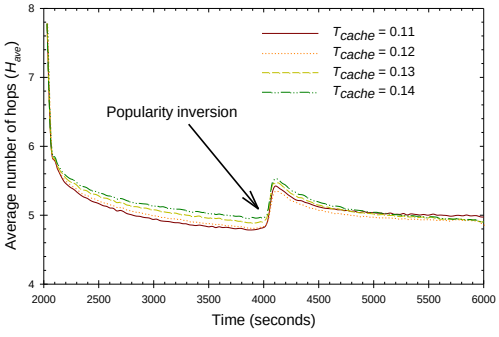

Fig. 8. Convergence of network after popularity inversion in community caching. Bucket size = 30 s, C_n = 20, θ = 0.1, hop_{max} = 4.

among nodes reflecting the ORT, while it has to be explicitly defined in [5], [6], [7]. When many nodes at higher levels of the ORT start caching popular keys, they do not forward messages to lower-level nodes. Therefore, the relative popularity of keys cached at lower-level nodes reduces. Consequently, cache storage allocated to those keys is reallocated to less popular keys. This is not possible in PoPCache and Beehive, as they force all the nodes along the ORT or within a specific address range to cache keys. Analytical and simulation results with varying Zipf's parameters (α) are also in good agreement confirming the correctness of the mode (Section A4.1). We also compare the performance against family of allocations derived from existing literature (Section A4.1). This analysis also justifies the allocation in Theorem 1 confirming slightly more cache capacity should be allocated for less popular keys (when query distribution is highly skewed).

6.2 Community-Based Caching

Fig. 5 compares H_{ave} of the entire system under *Chord*, *passive caching* (i.e., nodes cache responses to their own queries), *caching* (i.e., *LKDC* algorithm without sub-overlays), and the overall *community-caching* solution (i.e., *LKDC* deployed on top of sub-overlays). H_{ave} converges with all the caching schemes as the query distribution is steady. *Passive caching* reduces the H_{ave} from 7.79-hops to 7.32-hops and it is further reduced to 5.88 and 4.64 hops by *caching* and *community caching*, respectively. Thus, our caching solution reduces H_{ave} by 40.5%.

H_{ave} observed by each community is shown in Fig. 6. Communities with highly skewed query distributions (i.e., large α) and/or lower number of distinct keys gained significant performance improvement. For example, H_{ave} of communities m_3, m_7, and m_9 is reduced by 53%, 48%, and 48%, respectively. Moderately popular communities m_1, m_2, m_5, m_6, m_8, and m_{10} gained 31-42% improvement. Most popular query in m_4 had a global rank of 285 (Table 2); therefore, it gained only 7.4% improvement with *caching*. However, *community caching* was able to reduce the hop count by 23% (3.1 times improvement over *caching*). Performance gain by each community was dependent only on its popularity distribution, and both large and small communities benefited equally, e.g., {m_1, m_5 and m_6}, and {m_7 and m_9}. Performance under different numbers of communities and their relative size is analyzed in Section A4.2.1. These results show that performance gained by a community is independent of the number of communities in the system and even relatively small communities gain significant performance improvements compared to system-wide caching. These confirm the effectiveness of our sub-overlay formation mechanism to find community members. Sub-overlay formation can also reduce the latency of geographic communities by 33-50% (Section A4.2.3).

Performance gain with increasing C_n is shown in Fig. 7. Though H_{ave} rapidly reduces with increasing C_n, it tends to saturate after a while. This is an artifact of the Zipf's-like popularity distribution where significant performance can be gained by caching a few highly popular resources. Yet, diminishing return is gained with very large caches. Thus, while trying to provide a guaranteed mean, both Beehive and PoPCache had to force the nodes to cache several hundreds of resources on average and several thousands in the worst case, regardless of nodes'

capabilities or interests. In contrast, our caching scheme provides comparable lookup performance using small caches. Lookup performance under heterogeneous cache capacities is presented in Section A4.2.4. In Section A4.2.5, we also show that convergence time and overhead of the algorithm can be controlled using the weighting factor for query demand θ and caching threshold T_{cache}.

To observe the adaptability of our solution to rapid popularity changes, we invert the popularities of queries, where the least popular query suddenly becomes the most popular and vice versa. This is a worst-case scenario. Fig. 8 shows the convergence of the network after popularly inversion around 4,000 *s*. H_{ave} increased only by 0.5-hops and network stabilized following the same convergence pattern. It is sufficient to select hop_{max} = 4. We do not expect a significant increase in hop_{max}, even for a very large network, as it is inversely proportional to community size N_m. Our solution introduces minimum overhead as cache and member-discovery requests are piggybacked on *get()* and overlay maintenance messages, respectively. Caching alleviates hot spots within an overlay network because many nodes can answer popular queries. CBC solution was able to reduce the maximum number of queries answered by a Chord node from 25,151 to 1,677 (15 times reduction). Similarly, peak number of queries forwarded by a node was reduced from 27,574 to 5,191 (5.3 times reduction). Thus, the proposed solution also provides good load balancing properties. Detailed discussion on future work is presented in Section A5.

7 Conclusion

A sub-overlay formation and a distributed caching algorithm that adapts according to the interest patterns of explicit P2P communities are proposed. An analytical solution is used to determine the best cache placement and capacity allocation strategies and to provide useful bounds on performance. Caching algorithm utilizes only the local statistics, is adaptive, and works with any skewed distribution of queries. Overall solution enhances both the communitywide and systems-wide lookup performance, and introduces minimum storage, network, and computational overhead.

Acknowledgment

This research is supported in part by the Engineering Research Center program of the National Science Foundation under award number 0313747.

References

[1] B. Zhang, A. Iosup, J. Pouwelse, D. Epema, and H. Sips, "Sampling Bias in BitTorrent Measurements," *Proc. Euro-Par*, Aug.-Sep., 2010.

[2] S.B. Handurukande, A.M. Kermarrec, F.L. Fessant, L. Massoulié, and S. Patarin, "Peer Sharing Behaviour in the eDonkey Network, and Implications for the Design of Server-less File Sharing Systems," *Proc. EuroSys '06*, vol. 40, Apr. 2006.

[3] D. McLaughlin et al., "Short-Wavelength Technology and the Potential for Distributed Networks of Small Radar Systems," *Bull. Amer. Meteor. Soc.*, vol. 90, pp. 1797–1817, Dec. 2009.

[4] K. Sripanidkulchai, "The Popularity of Gnutella Queries and its Implications on Scalability," http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html, Feb. 2001.

[5] V. Ramasubramanian and E.G. Sirer, "Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays," *Proc. USENIX NSDI '04*, vol. 1, pp. 99-112, 2004.

[6] W. Rao, L. Chen, A.W. Fu, and Y. Bu, "Optimal Proactive Caching in Peer-to-Peer Network: Analysis and Application," Proc. *6th ACM Conf. on Information and Knowledge Management*, pp. 663-672, Nov. 2007.

[7] W. Rao, L. Chen, A.W.-C. Fu, and G. Wang, "Optimal Resource Placement in Structured Peer-to-Peer Networks," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 21, no. 7, July 2010.

[8] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The Impact of DHT Routing Geometry on Resilience and Proximity," *Proc. ACM SIGCOMM '03*, pp. 381-394, Aug. 2003.

[9] E. Cohen and S. Shenker, "Replication Strategies in Unstructured Peer-to-Peer Networks," *Proc. ACM SIGCOMM '02*, pp. 177-190, Aug. 2002.

[10] S.A. Theotokis and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Comput. Surveys*, vol. 36, no. 4, pp. 335-371, Dec. 2004.

[11] L. Liu, J. Xu, D. Russell, and Z. Luo, "Evolution of Social Models in Peer-to-Peer Networking: Towards Self-Organising Networks," *Proc. 6th Int. Conf. on Fuzzy Systems and Knowledge Discovery*, 2009.

[12] F. Xue, G. Feng, and Y. Zhang, "CommuSearch: Small-World Based Semantic Search Architecture in P2P Networks," *Proc. IEEE GLOBECOM* '10, Dec. 2010.

[13] M. Bertier, D. Frey, R. Guerraoui, A. Kermarrec, and V. Leroy, "The Gossple Anonymous Social Network," *Proc. ACM/IFIP/USENIX 11th Middleware Conf.*, pp. 191-211, Dec. 2010.

[14] N. Zeilemaker, M. Capotă, A. Bakker, and J. Pouwelse, "Tribler: P2P Media Search and Sharing," *Proc. 19th ACM Int. conf. on Multimedia*, pp. 739-742, Nov.-Dec. 2011.

[15] I. Stoica et al., "Chord: A Scalable Peer-to-Peer Protocol for Internet Applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17-32, Feb. 2003.

[16] H.M.N.D. Bandara and A.P. Jayasumana, "Exploiting Communities for Enhancing Lookup Performance in Structured P2P Systems," *Proc. IEEE ICC '11*, June 2011.

[17] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed, "Magnet: Practical Subscription Clustering for Internet-Scale Publish/Subscribe," *Proc. 4th ACM Int. Conf. on Distributed Event-Based Systems*, July 2010.

[18] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," *Proc. Int. Workshop on Peer-To-Peer Systems*, 2003.

[19] Z. Despotovic, Q. Hofstätter, M. Michel, and W. Kellerer,, "An Operator Approach to Popularity-based Caching in DHTs," *Proc. Int. Conf. on Communications (ICC '10)*, May 2010.

[20] H.M.N.D. Bandara, "Enhancing Collaborative Peer-to-Peer Systems Using Resource Aggregation and Caching: A Real-World, Multi-Attribute Resource and Query Aware Approach," *PhD Dissertation*, Colorado State University, Fall 2012.

[21] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," *Proc. 10th Global Internet Symp.*, pp. 79-84, May 2007.

H. M. N. Dilum Bandara received B.Sc. (First Class Honors) in Computer Science and Engineering from University of Moratuwa, Sri Lanka in 2004 and M.S. in Electrical and Computer Engineering from Colorado State University in 2008. He is currently a Ph.D. candidate at Colorado State University. He is a lecturer at Dept. of Computer Science and Engineering, University of Moratuwa, Sri Lanka since 2005 (currently on study leave). He is a student member of the IEEE.

Anura P. Jayasumana received B.Sc. in Electronic and Telecommunications Engineering, with First Class Honors from University of Moratuwa, Sri Lanka and M.S. and Ph.D. in Electrical Engineering from Michigan State University. He is a Professor in Electrical and Computer Engineering at Colorado State University, Fort Collins. He is a member of the NSF Engineering Research Center for Collaborative Adaptive Sensing of Atmosphere (CASA). He is a member of Phi Kappa Phi and a senior member of the IEEE.