Computer Networks 99 (2016) 37–50

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Shades: Expediting Kademlia's lookup process

Gil Einziger[1], Roy Friedman*, Yoav Kantor

Computer Science Department, Technion, Haifa 32000, Israel

ARTICLE INFO

Article history:
Received 17 September 2015
Revised 28 January 2016
Accepted 29 January 2016
Available online 12 February 2016

Keywords:
Kademlia
Peer-to-peer
Lookup process
Caching

ABSTRACT

Kademlia is considered to be one of the most effective key based routing protocols. It is nowadays implemented in many file sharing peer-to-peer networks such as BitTorrent, KAD, and Gnutella. This paper introduces *Shades*, a combined routing/caching scheme that significantly shortens the average lookup process in Kademlia and improves its load handling. The paper also includes an extensive performance study demonstrating the benefits of Shades and compares it to other suggested alternatives using both synthetic workloads and traces from YouTube and Wikipedia.

1. Introduction

Distributed Hash Tables (DHT) are at the heart of most peer-to-peer (P2P) systems. Consequently, a plethora of papers and ideas on how to implement DHTs have been published, e.g., [4,34]. DHTs tend to differ from each other in the routing scheme they employ, as well as the space and message overhead they incur for maintaining their respective overlays. During the last few years, Kademlia has become one of the most widely used DHTs in practice [36,39]. This is largely due to its proven robustness to churn, enabled by its unique partially parallel lookup mechanism and large routing tables.

Further, Kademlia's applications extend beyond P2P. For example, a variant of Kademlia was suggested for high performance computing in Grids and clusters [43].

Like many other DHTs, Kademlia's routing phase may involve contacting a logarithmic number of nodes, which may be too slow for time sensitive applications [32,37]. For example, one of the lessons of the CoralCDN project [22], a successful DHT based content delivery network, is that DHT lookup latency was a performance bottleneck for their system [21].

Since typical workloads of Internet based applications are often highly skewed, caching lookup results along the search path has the potential of reducing the average lookup time experienced by users. However, due to Kademlia's unique routing and dynamic bucket manipulation schemes, caching is less effective in Kademlia than in more rigid DHTs like Chord [16].

1.1. Contribution

We introduce Shades, a novel caching and augmented routing mechanism for Kademlia that reduces the number of nodes participating in the lookup process and lowers the load of the most congested nodes. Such nodes are identified as a performance bottleneck for DHTs in general [29,33] and for Kademlia [8] in particular.

In Shades, nodes are equipped with a small local cache that can shorten lookups in case of a cache hit. Shades exploits a secondary hashing scheme, called *colors*, that makes the caches more specialized. Thus, each cache receives more traffic for items of its own color, thereby increasing cache hit rates. This way, Shades achieves a reduction in the number of contacted nodes per lookup when compared to previous caching schemes.

* Corresponding author.
E-mail addresses: gmail1983@gmail.com (G. Einziger), roy@cs.technion.ac.il (R. Friedman), ykantor@cs.technion.ac.il (Y. Kantor).

[1] Partially supported by the Technion HPI Research School.

http://dx.doi.org/10.1016/j.comnet.2016.01.014
1389-1286/

We have experimented with Shades and compared it to plain Kademlia and other previous caching suggestions for Kademlia, namely *KadCache* – the caching suggestion of the Kademlia authors [36], the local cache suggested in [24] – a.k.a. *Local* and *Kaleidoscope* [16]. These experiments were conducted using both synthetic workloads mimicking ones that are often found in real applications, as well as real traces from YouTube and Wikipedia. In these results, we have found that Shades significantly reduces the number of nodes participating in the lookup process compared to plain Kademlia, KadCache, Local and Kaleidoscope. At the same time, it also achieves competitive message and bandwidth overheads relative to the other suggested caching schemes.

The rest of this paper is organized as follows: We start by describing Kademlia in Section 2. Section 3 surveys additional related work. Shades is presented in Section 4 and is analyzed in Section 5. The performance evaluation appears in Section 6. Finally, we conclude with a discussion in Section 7.

2. A brief overview of Kademlia

Kademlia is described in [36]. Here we only give a brief overview of its structure and main properties as a background to our work. In Kademlia, each node and each object are assigned a 160-bit key using a hashing function (such as SHA-1). The notion of distance is defined in Kademlia using the XOR metric and objects are stored in the nodes whose id is closest to theirs according to this metric.

In Kademlia, each node maintains a *bucket* of up to k nodes (or *k-bucket*) for each of the 160 bits of its key. The k-bucket corresponding to the ith bit of node p stores up to k nodes whose distance from p is between 2^i and 2^{i+1}. This overlay may constantly evolve as p learns about additional nodes. Each k-bucket is kept sorted by the least-recently seen order. Each time p receives a message from q that corresponds to a given k-bucket of p, if q does not exist there and the bucket is not full, it is added to the bucket as the most-recently seen node. If q already exists in the bucket, it is simply updated to be the most-recently seen node there. Finally, if the bucket is full and q is not there, then the least-recently seen node r in the bucket is pinged. If r answers, then it is updated to be the most-recently seen node. Otherwise, q replaces r in the k-bucket and is marked as the most-recently seen node.

The routing process of Kademlia proceeds as follows: When a node p needs to find the node (or value) with key d, p begins the following iterative partially parallel lookup process. p creates a list of the k closest nodes to d according to its k-buckets (possibly including itself); call this list the *k-candidate list* for d. It is possible that this list contains fewer than k nodes. Then, on each iteration, p picks the first α (a parameter greater than 0) nodes in the k-candidates list that were not queried yet and sends each of them in parallel an asynchronous query for d. In response, each of these nodes r returns to p the k closest nodes to d according to r's k-buckets. Following the reply, p updates its k-candidates list to hold the k closest nodes to d that p knows about thus far. Also, after each such reply, or a time-out on a given recipient, p sends a query to the first node in its k-candidates list that it has not contacted yet (having up to α outstanding queries at any given time). This process ends when all nodes in the k-candidates list have been queried. At this point, the nodes in the k-candidate list are declared as the k closest nodes to d.

Similar to many DHTs, it has been proved in [36] that the lookup time of Kademlia is at most logarithmic. In contrast with some other well known DHTs, such as Chord [38], the Kademlia overlay constantly evolves, yet need not be updated immediately following joining or departure of nodes. Also, each entry has k nodes, giving the routing protocol some freedom in choosing its preferred route. Hence, two consecutive lookup requests for the same item may follow different routes even if there is no churn in the network.

3. Related work

3.1. Shorter search in DHTs

Several works have investigated how to use caching to reduce the lookup length in DHTs. For example, in [24] it is suggested to add to Kademlia a local cache named *Fast Table*. This table stores the results of previous lookups the node has performed. When a node receives a find value request, it first checks its Fast Table to see if it contains cached results for it. This approach was shown in [16,24] to yield a reduction in average lookup length. As mentioned in the introduction, we refer to this scheme as Local in this paper.

Another important caching suggestion appears in the original Kademlia paper [36]. In this suggestion, every time a node performs a find value operation, it sends a store value request to the last node it contacted that did not have the value. This suggestion, called KadCache in this paper, was evaluated in [16] for its message cost and (lack) of load balance capabilities. In this paper, we extend that evaluation of KadCache to cover its average and median query length. As we show in the performance section of this paper, Shades reduces considerably the number of contacted nodes compared to both Local and KadCache, and usually also improves the communication overhead.

The work most related to Shades is Kaleidoscope [16]. Kaleidoscope also uses colors to augment the combined routing and caching process of Kademlia to obtain better caching, but focuses on communication overhead reduction. In Kaleidoscope, messages are first forwarded to a node of a matching color along the lookup path, and only then an iterative lookup starts. Since Kaleidoscope never deviates from the lookup path, it cannot efficiently use as many colors as Shades, and therefore achieves lower cache hit rates. Further, the more colors Kaleidoscope uses, the longer it take to reach each cache.

Unlike Kaleidoscope, Shades may deviate from the lookup path of Kademlia if there is probabilistic evidence that doing so is likely to find a cached result nearby. Shades bases its decisions on a compressed approximated statistics in order to both manage its cached content, and also decide on the maximal number of cache lookups that may deviate from the Kademlia lookup path. So while both

Table 1
Comparison between Kaleidoscope and Shades.

	Kaleidoscope	Shades
# Colors	17	150
On path lookups	Unlimited	Unlimited
Deviates from path	No	Yes
Time of first cache lookup	During lookup	First step
Cache policy	LRU	LazyEvict+TinyLFU
Share policy	Always	Only if needed

Kaleidoscope and Shades rely on the notion of colors as a secondary hashing mechanism, each takes this concept in a completely different direction.

The main differences between Kaleidoscope and Shades are summarized in Table 1. As can be seen, Shades uses more colors than Kaleidoscope and therefore forms a more effective distributed cache. Further, Shades benefits more from each cache hit as it performs the first cache lookup earlier than Kaleidoscope. Shades also uses a more advanced cache policy that is also used to decide how many times we deviate from the lookup path, and what node is most suitable to store the cached value at the end of the lookup. Finally, the last line of the table titled "share policy" indicates that shades stores the results of successful lookups in caches of matching colors that were encountered along the lookup process only if these caches are likely to benefit from them. In contrast, Kaleidoscope always pushes the results of lookups to such caches. This helps Shades save messages. Evidently, in our performance evaluation section, we show that Shades contacts substantially fewer nodes than Kaleidoscope, obtains significantly better load sharing, and generates similar overall traffic as Kaleidoscope.

Other methods to reduce Kademlia's lookup latency includes careful parameter configuration [39], techniques to fill k-buckets with nodes of geographical proximity [9,30], a new metric based on geographical distance [23,35,41] and a recursive lookup scheme [27]. We believe that many of these suggestions can be deployed alongside with Shades as they either reduce the latency of individual messages, or optimize the configuration parameters of the protocol. In contrast, Shades slightly changes the algorithm and satisfies lookups using information from fewer nodes.

As for non-Kademlia DHTs, multiple works have explored caching in structured overlays such as Chord and Pastry, e.g., [5,12,13] to name a few. Caching in such overlays is likely to be effective, at least when the churn rate is low, since search paths are deterministic and stable. Thus, once a popular item is cached in a node along a search path, it will likely result in multiple cache hits by future searches for the same item. As we explained above, this is not the case in Kademlia, which motivated our study.

Other DHT's like OneHop [25], Kelips [26] and Tulip [3] achieve O(1) lookups at the cost of background traffic overheads. In contrast, Shades does not generate any background traffic. Systems that provide O(1) lookups include, e.g., Dynamo [14] and ZHT [31]. Both systems target high performance data centers. Given that a variant of Kademlia was also suggested for this context [43], Shades can also be adopted to that domain.

Fig. 1. A counting Bloom filter.

3.2. Brief survey of approximate counting techniques

Since TinyLFU uses approximate counting techniques, we provide a brief survey of approximate counting below. The survey helps in understanding the internal structure of TinyLFU in Section 4.1.2.

Bloom filters [6] are space efficient approximated data structures for answering set membership queries. Bloom filters support two methods: ADD and CONTAIN. A Bloom filter uses k hash functions, h_1, h_2,..., h_k to hash elements over an array of m bits. When adding an element T, $h_1(T)$, $h_2(T)$,..., $h_k(T)$ are calculated and the matching bits of the array are set to 1.

The CONTAIN method hashes the item and tests the appropriate bits; if all the bits are set, the CONTAIN method returns true. If the Bloom filter is properly configured, this answer is usually correct. Yet, if one of the bits is unset, the item is not contained in the set (always). We call the case when the CONTAIN method inaccurately includes an element in the set a *false positive*. The false positive probability of a Bloom Filter that contains N elements is $(1-(1-\frac{1}{m})^{kN})^k \approx (1-e^{-kN/m})^k$.

While Bloom filters are able to store a set, they cannot count how many times an item was inserted. *Counting Bloom filters* (CBF) [19] are a natural extension of Bloom filters that support multiplicity queries and deletions. To do so, a counting Bloom filter replaces the bit array with a counter array. That is, instead of setting k bits (1 per hash function), the counting Bloom filter increments k counters.

The contain operation simply checks whether $h_1(T)$,..., $h_k(T)$ are all greater than zero. Counting Bloom filters typically also support a *remove* operation that calculates $h_1(T)$,..., $h_k(T)$ and decrements the corresponding counters in the counter array. This operation is however not useful in our context. Instead, we are interested in answering *multiplicity queries*. That is, for an item T, we would like to know how many times T was inserted to the CBF. To do so, $h_1(T)$, $h_2(T)$,..., $h_k(T)$ are calculated, and the corresponding counters are read. The minimal value of the counters is returned as the multiplicity estimation of T. This operation is described in detail in [11].

Since only the minimal value of the corresponding counters ($h_1(T)$,..., $h_k(T)$) determines the multiplicity estimation of the CBF, a very efficient optimization is to increment only the counters whose value is the minimal. Intuitively, this optimization prevents low frequency items from influencing counters that are associated with high frequency items. This optimization is called *Minimal Increment* in [11] and *Conservative Update* in [17]. It was shown to significantly increase the accuracy of high frequency items.

Fig. 1 illustrates a counting Bloom filter with 3 hash functions. In this examples, if we assume that the marked counters are the corresponding counters of an item (T), then the multiplicity estimation of T is 2 since this is the

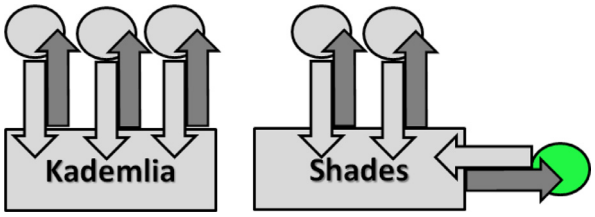


Fig. 2. A high level overview of the lookup process in Shades and Kademlia. As illustrated to the left, in Kademlia, ($\alpha = 3$) parallel messages are sent according to the Kademlia protocol (that uses the XOR metric). In Shades, two messages follow the original protocol and the third follows a new protocol. That is, the third message is sent to a node that has the same color as the searched key. The cache of that node is more likely to contain the searched value and with high enough cache hit rate the lookup process is reduced.

minimal corresponding counter value. Further, adding T to the CBF will increment all 3 counters, or only the two left counters if the minimal increment optimization is used.

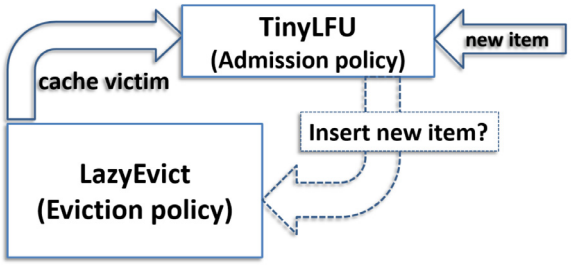
4. Shades

The idea behind Shades is to employ two lookup mechanisms in parallel: one mechanism to ensure correctness and termination and the other one to improve average case performance. That is, Shades uses Kademlia's original routing scheme and at the same time performs cache lookups that may conclude the lookup earlier in case of a cache hit. This idea is illustrated in Fig. 2. As can be observed, Kademlia routes several parallel lookups with the same (XOR) metric while Shades uses Kademlia's routing technique as well as an additional technique that amplifies the chances of hitting a cache early in the search.

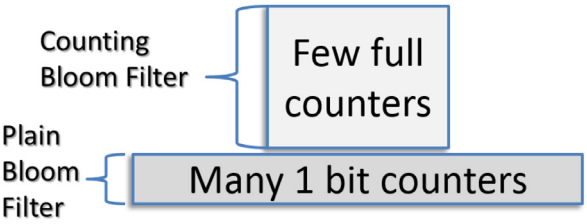
Specifically, in Shades, each node includes a small local cache that is managed with an effective cache admission scheme called *TinyLFU* [15]. TinyLFU maintains frequency approximation of recently encountered items that gradually adopt to changes in the access distribution. In TinyLFU, a new item is only admitted to the cache if it is estimated to be more frequent than the cache victim. Therefore, TinyLFU only admits popular items into caches, and the goal of Shades is to make different caches see different items as popular.

To do that, Shades uses a secondary key called *color*, which is derived from the Kademlia key, as described below. Shades routes some of the requests to nodes whose color matches that of the lookup. Therefore, caches are more likely to be searched according to color, which increases the cache hit rate. Shades also uses the frequency estimations of TinyLFU to decide how many times it is worthwhile to use the color routing technique and deviate from the original Kademlia protocol. This approach significantly reduces the overheads associated with Shades.

In summary, Shades includes three components: a highly effective small cache, an augmented routing that is based on secondary hashing (colors) whose goal is to direct lookup traffic to caches that are likely to have the data, and an overload protection mechanism. The caching mechanism is described below in Section 4.1, the routing scheme is presented in Section 4.2, and the overload protection is explained in Section 4.3.



(a) TinyLFU integrated with LazyEvict.



(b) TinyLFU's internal structure

Fig. 3. An illustration of TinyLFU integration with caches and its internal structure.

4.1. Caching mechanism

With Shades, each node in the system has a small local cache in addition to its Kademlia storage. When a node receives a find value request, it can either return the k-closest nodes, return a cached result or return the stored value. The Kademlia storage stores every key/value pair it is assigned to by Kademlia for correctness, while the cache stores additional key/value pairs to improve performance. Ideally, we would like the cache to be small and include the most frequently requested key/value pairs.

4.1.1. Single cache management

For the cache management, we employ the general cache architecture of TinyLFU [15] where there is a clear separation between the cache eviction policy and its admission policy. The eviction policy is responsible for picking a cache victim, while the admission policy decides whether admitting the new item at the expense of the cache victim is beneficial to the cache.

Fig. 3(a) illustrates the operation of TinyLFU. TinyLFU stores statistics about the frequency of recently encountered items. These statistics are used in order to estimate the recent frequency of both the cache victim and the newly arriving item. The latter will only be admitted to the cache if it is estimated to be more frequent than the cache victim.

4.1.2. TinyLFU internal structure and operation

Maintaining an approximate statistics can be done, e.g., with a counting Bloom filter employing the minimal increment optimization, as described in Section 3.2. The

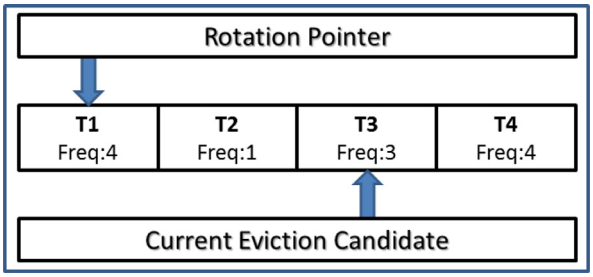


Fig. 4. An illustration of a cache augmented by LazyEvict eviction policy, and the manner the cache victim is picked.

number of (multiple bits) counters needed depends on the number of items the counting Bloom filter is supposed to represent. However, most real world workloads exhibit highly skewed access distributions, meaning that most items are accessed very rarely. This results in a non-negligible waste since their corresponding counters are likely to contain either 0 or 1.

TinyLFU solves this issue by adding a (plain) Bloom filter, nicknamed *doorkeeper*, in front of the main counting Bloom filter, as illustrated in Fig. 3(b). Only items that already exist in the doorkeeper are added (when accessed) to the main counting Bloom filter. Consequently, the latter can contain considerably fewer counters than would be needed in a naive implementation. TinyLFU also includes an aging mechanism, which continuously adapts the frequency estimation to the observed access distribution, and is highly space efficient [15]. As shown in [15], the computational complexity of queries and updates in TinyLFU is $O(1)$. TinyLFU also perform a periodic aging operation on its counters, whose deamortized cost is also $O(1)$.

4.1.3. LazyEvict

As reported in [15], once employing an admission policy such as TinyLFU, the impact of the eviction policy becomes almost marginal. In fact, for skewed workloads like the ones we experience here, even naive eviction policies result in close to optimal ratios between the cache size and its hit rate.

Thus, we aim for a replacement policy whose maintenance cost is very low. To that end, we have developed the *LazyEvict* replacement policy, which aspires to eventually return the least frequently accessed object as the eviction candidate, but searches for it lazily, advancing one object at a time. Specifically, LazyEvict employs two pointers whose maintenance and query complexity is $O(1)$ (in addition to the cost of TinyLFU). One pointer points to the current eviction candidate and the other is used to rotate among all cached items, as outlined in Fig. 4. On each access, the rotation pointer is advanced by one and the approximated access frequency of the corresponding object as maintained by TinyLFU is compared to that of the current eviction candidate. If the current eviction candidate is more frequent, then the eviction candidate pointer will now point to the same item as the rotation pointer.

In the example of Fig. 4, during the first insertion, T1 will be compared against the current eviction candidate (T3). Since its frequency is larger than the eviction candidate's, LazyEvict will not replace them and TinyLFU will compare between the newly arriving object and T3. During the next access, T2 is compared against the eviction candidate (T3). Since its frequency is lower than that of T3, T2 will be pointed to by the eviction candidate pointer and TinyLFU will then compare the arriving item against the updated eviction candidate (T2).

Further, in order to avoid using the TinyLFU histogram multiple times, we maintain the frequencies of in-cache items explicitly, effectively having to use TinyLFU only to estimate the frequencies of items that are not presently stored in the cache.

The main benefits of LazyEvict are simplicity and practicality. LazyEvict is cache friendly since items are scanned one after another and we do not change the position of items in the cache (or its meta-data) once admitted. We also do not have contention over the head of the list like LRU does [18]. Our measurements reported in Section 6.3 below show that for our target workloads, TinyLFU+LazyEvict yields results very similar to a true LFU cache, which is a lot more complex to implement as it requires keeping the in-cache items ordered at all times, and therefore uses heaps whose access complexity is $O(\log(N))$. We therefore chose to use TinyLFU as it has better complexity than the heap implementation and provides almost the same hit rate.

4.2. Routing

4.2.1. Colors

As mentioned before, Shades augments the standard Kademlia routing scheme by utilizing a secondary key called *color*. The color is generated by applying a hash function to the Kademlia key. Unlike the Kademlia key that comes from a large domain to prevent collisions, the color domain is small and collisions are desired. Recall that shades relies on completely autonomous TinyLFU managed caches. That is, the TinyLFU policy of each cache separately tracks the observed access histogram and each cache stores the most frequent items it has encountered. The purpose of Shades' routing protocol is to ensure that each node receives a disproportionate amount of requests for its specific color, and therefore its cache is more likely to admit items of that color.

During the parallel iterative lookup process, Shades may issue cache lookup requests to nodes that have the same color as the color of the requested key even if these nodes do not advance in the XOR metric. For this reason, we call such deviations *side steps*. Hence, while intuitively a side step improves the chances of hitting a cache due to the use of colors, in the case of a cache miss it prolongs the lookup process since it does not advance toward the key in the XOR metric. In order to avoid paying this price for cache misses, Shades only takes side steps if the item is relatively likely to be cached already. To that end, Shades relies on TinyLFU to keep track of the likelihood that the item would indeed be in the cache, as detailed later in Section 4.2.

Finally, once the lookup is done, the search result is only stored in caches that are interested in caching it. Since TinyLFU only admits items to the cache if they are more

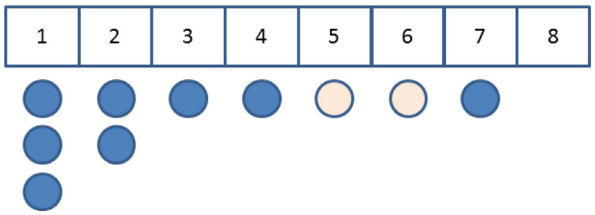


Fig. 5. Shades Palette.

frequent then the cached items, TinyLFU ensures that the cached result reaches a node that is likely to benefit from it. Below, we first describe an auxiliary data structure used by the routing mechanism of Shades and then provide the details of the protocol.

4.2.2. Palette

Since each Kademlia node typically remembers a large number of other nodes (up to a thousand), going over all the k-buckets in order to find a matching color candidate can be time consuming. Hence, each node p maintains a mapping between colors and the nodes matching these colors that p is aware of. This mapping, implemented as a hash table, is called the *Palette* of node p. For each color i, when node p has at least one node of color i in any of its k-buckets, then the ith entry of p's Palette points to these nodes. However, if p does not have any node of color i in any of its k-buckets, then we fill the corresponding entry with other nodes that p detects using the following pull gossip mechanism.

Whenever p sends a lookup message, it piggybacks on the lookup message a bitmap that represents which colors have no representatives in its Palette. I.e., bit i in the bitmap contains 1 if p is already aware of at least one node of color i and 0 otherwise. When a node q receives such a lookup message, it piggybacks on the reply one node corresponding to the color of each 0 bit in the bitmap that q is aware of (if q knows such a node). In addition, q includes at least one node whose color matches the color of the searched key. All this data is piggybacked on existing messages to avoid generating new messages. The size of piggybacked data is relatively small: a bitmap whose size in bits is the number of colors and at most one id per color (and typically only a few ids or none at all).

Shades' Palette is illustrated in Fig. 5. In this example, there are 8 different colors. The dark tokens represent the nodes that appear in the k-buckets whereas the bright tokens are nodes discovered through the bitmap gossip mechanism. In this example, color 8 does not have any representative. Therefore the bitmap [11111110] will be added to any outgoing Kademlia message. If any of the nodes that receive such a message knows of a node that matches color 8, it will include this node in its response.

4.2.3. Shades routing protocol

The routing protocol for key lookup, performed by node p, goes as follows. Denote c the searched key's color. While node p is not aware of c-colored nodes, p performs traditional Kademlia lookups. When node p is aware of c-colored nodes, either from its data structures or through replies received from other nodes, it performs multiple cache lookups denoted as *side steps*. These cache lookups are performed simultaneously to Kademlia's routing protocol. As mentioned before, side steps do not necessarily advance the search according to the Kademlia XOR metric.

Let q be the c-colored node that is closest to the searched key. The first side step is performed by sending a request to node q. q does not have to be in the k-candidates list. q checks whether the requested key is in its cache. If so, it sends back the (key, value) item from the cache. Otherwise, q returns a response that contains the following additional information:

- Is the item needed? i.e., will this specific cache admit this item if encountered based on the mechanism described in Section 4.1.
- Is the item popular? i.e., is this item popular enough to be likely admitted to other caches.

The "is needed" bit is set if the arriving item would be admitted to this node cache. The "is popular" bit is set if TinyLFU gives this item a score that is greater than a predefined threshold (1 in our implementation). In that case, since the item was recently encountered it is possible that other caches (of matching color) store this item, as each node measures a slightly different access frequency.

When p receives the response from q, it acts according to the response: In case of a cache hit, the lookup is finished. Otherwise, if the item is not popular, then no more side steps are performed and the lookup is continued as in Kademlia. If the item is popular, then another side step can be taken. Note that by this point, p received more c-colored nodes from responding nodes. If p discovered more than one c-colored node, it favors contacting the closest one according to the XOR metric.

At the end of the lookup, if the lookup is successful, p sends the (key,value) item to the c-colored node that is closest to the searched key and has noted in its response that the value is needed. This node stores the result in the cache for future requests.

Shades, as Kademlia, has up to α outstanding queries at any given time. When not performing a side step, all the outstanding queries advance according to the key XOR distance metric as in Kademlia. While performing a side step, $\alpha - 1$ of the outstanding queries advance according to the key distance metric in addition to the one outstanding side step.

Note that in order to perform a side step, p needs to know a node with the same color as the searched key. Recall that the Palette significantly increases the probability that p knows such a node. This enables our routing protocol to usually perform the first side step right in the beginning of a lookup, which is important since the benefit of hitting a cache early is far greater than hitting it later.

Algorithms 1–3 describe Shades' routing protocol. The variables declared in Algorithm 1 in the global scope are accessible to all three algorithms. Algorithm 1 describes the way the request messages are sent. Algorithm 2 describes the way the request messages are handled by nodes that receive them and the information sent in their responses. Algorithm 3

Algorithm 1 Find Value: Sending Requests.

```
1: boolean isRequestPopular = true
2: boolean isThereOutstandingSideStep = false
3: Node sideStepNode = null
4: int numberOfSidesteps= 0
5: Nodes kCandidates= null
6: Nodes coloredNodes= null
7:
8: function FINDVALUEREQUEST(Key key)
9:     if myCache.HasItem(key) then
10:        return myCache.GetItem(key)
11:    kCandidates = closest k nodes found in k-buckets
12:    coloredNodes = nodes having the same color as the searched key's color from the Palette
13:    Do the following with alpha concurrency:
14:    while true do
15:        if all k-candidates are queried then
16:            return k-candidates
17:        if shouldDoSideStep() then
18:            doSideStep()
19:        else
20:            doKademliaStep()
21:
22: function SHOULDDOSIDESTEP( )
23:    if ((isRequestPopular==true) AND (isThereOutstandingSideStep==false) then
24:        isThereOutstandingSideStep= true
25:        return true
26:    else
27:        return false
28:
29: function DOSIDESTEP( )
30:    Node node = the closest unqueried node from coloredNodes
31:    ▷ If a sidestep was already taken, node must be closer to the searched key then the previous queried node
32:    sideStepNode = node;
33:    Send node a request attached with palette.getBitmap()
34:
35: function DOKADEMLIASTEP( )
36:    Node node = get the closest unqueried node from the k-candidates list
37:    Send node a request attached with palette.getBitmap()
```

Algorithm 2 Find Value: Handling Requests.

```
1: function FINDVALUEHANDLE(findValueRequest request)
2:    Key key = request.getKey()
3:    FindValueResponse response= new FindValueResponse()
4:    response.setColorNodeAsKey( getColorNode(key))
5:    response.setColorNodes(getNodesByBitmap(request.getBitmap()))
6:    if myCache.hasItem(key) then
7:        response.setItem( myCache.getItem(key))
8:        response.setCacheHit(true)
9:    else
10:       response.setKCandidates(k-candidates from the Palette (that appear in k-buckets)
11:       if key.getColor()==myColor then
12:           response.setPopularity( myCache.isItemPopular(key))
13:           response.setIsNeeded( myCache.isItemNeeded(key))
14:   Send response
15:
16: function GETCOLORNODE(Key key)
17:    return closest node with the same color as key from the Palette
18:
19: function GETNODESBYBITMAP(Bitmap bitmap)
20:    return up to k nodes missing according to the bitmap
```

Algorithm 3 Find Value: Receiving Responses.

```
1: function FINDVALUERESPONSE(findValueResponse response)
2:    palette.update( response.getBitmapNodes())
3:    coloredNodes.add( response.getColorNodeAsKey())
4:    if response.isCacheHit() then
5:        return response.getItem()
6:    kCandidates.update(response.getKCandidates())
7:    if response.getSender()==sideStepNode then
8:        isThereOutstandingSideStep= false
9:        isRequestPopular= response.isPopular()
10:       if response.isNeeded() then
11:           nodesThatNeedTheItem.add(sideStepNode)  ▷ When/if the item is found, it is sent to the closest node in nodesThatNeedTheItem
```

describes the way the requesting node handles these responses and the way it updates its data structures. Note that the updated data structure in Algorithm 3 affects the subsequent requests in Algorithm 1.

4.3. Explicit Congestion Notification

When we started experimenting with Kademlia in general and with Shades in particular, we encountered a congestion problem that caused message drops. In order to avoid this problem, we have added a simple congestion control mechanism, inspired by the *Explicit Congestion Notification* of TCP/IP [20]. That is, every time a node handles a message while its incoming message queue is almost full (75% full in our measurements), this node marks this message with a congested bit. Nodes that receive messages marked with the congestion bit may replace congested nodes without a ping, similarly to the way Kademlia treats nodes that failed to respond to a message.

This mechanism helps Kademlia nodes avoid becoming overloaded with requests by reducing their frequency in routing tables. As a result, the most congested nodes receive fewer messages as can be seen in Section 6.9 below. We note that we added this optimization to all the algorithms and not just to Shades.

5. Analysis

Throughout the probabilistic analysis below, we make several simplifying assumptions as detailed below. Yet, as demonstrated by our performance evaluation in Section 6, the actual results follow closely our analysis, suggesting that the overall analysis is indicative despite these assumptions.

5.1. Simplifying assumptions

Assumption 1. All nodes request approximately the same number of values at an arbitrary common distribution.

Assumption 2. Keys are colored uniformly at random.

Assumption 3. A side step is performed during every lookup. (That is, a node with identical color to the searched item is accessed.)

5.2. Augmented popularity for same color items

In this section, we provide mathematical intuition to the claim that nodes observe augmented frequencies for items of their own color. Denote P_d the *popularity of a key d*, i.e., the probability of a given node to issue a request for *d*.

Denote C the number of colors and n the number of nodes. Since every node has a color, on average n/C nodes belong to each color. Therefore, since we assume that every lookup performs a sidestep, one of these n/C nodes with identical color to d is accessed during the lookup process. Since all lookups for d from n nodes contact a subset of the nodes of size n/C, any of these nodes are expected to get C times more requests for d than a random node in the system. That is, they experience a frequency of approximately $P_d\dot{C}$ requests for d, generated both locally and as a result of side steps and lookups.

If we neglect the routing traffic for key d for nodes that are not close to d in the XOR metric, these nodes receive just local requests for d, and observe a frequency of P_d.

For example, consider a value of popularity 1% ($P_d = 0.01$), and assume that each node issues a request just for a single item. Matching colored nodes see $C \cdot 0.01 = 150 \cdot 0.01 = 1.5$ requests for d in expectation while other nodes only see 0.01 requests for d in expectation.

5.3. Constant distribution hit rate bounds

We now characterize the performance envelope of Shades. In particular, we wish to explore the achievable hit rates in Shades under ideal conditions.

Assuming C colors and local caches of size S, the maximum distributed cache size is $C \cdot S$. The best hit rate is achieved when every cache contains only the most frequent items of a specific color and these items are distributed perfectly to colors. In this case, the maximal hit rate achievable by Shades (after warmup) is: $\sum_{i=0}^{C \cdot S} P_i$ where P_i is the relative frequency of the i'th item in the distribution.

Similarly, the worst case for Shades is when all the caches greedily cache the S most frequent items (regardless of color). In this case, all caches will cache the same S items. The "lower bound" hit rate is therefore $\sum_{i=0}^{S} P_i$. We show in the result section that the hit rate estimations remain indicative to the performance of the system, even though our LFU policy is approximated and may fail to identify the most frequent items, and despite our simplifying assumptions.

5.4. Load distribution between the colors

Ideally, all nodes should receive a similar load regardless of their color. Yet, as the request distribution is not necessarily uniform, the load experienced by nodes of different color is not equal. Below, we estimate the likelihood for deviation from the average expected load experienced by the nodes of a given color.

We associate each key d with a random variable X_d whose popularity is taken from the original distribution. Hence, $E(X_d) = P_d$. Using these random variables, we define new random variables $\{C_i\}$ to be the summation of all random variables of the color i. An immediate observation is that for any two colors i, j, $E(C_i) = E(C_j)$. This is because for every splitting that favors one of the colors over the other, we can change the color name and receive the opposite one. Let T be the total number of find value requests in the system so far. Hence, $E(C_i) = E(C_j) = \frac{T}{C}$ for every i, $j \in [1, C]$. By applying Markov's inequality, we conclude that:

$P(C_i > \beta E(C_i)) < \frac{1}{\beta}$ and therefore: $P(C_i > \beta \frac{T}{C}) < \frac{1}{\beta}$. This result limits the probability that a certain color is significantly more popular than other colors. In particular, the probability that a specific color is β times more popular than the average is less than $\frac{1}{\beta}$. To guarantee that this property holds for any key popularity distribution, only uniformly random hash functions should be employed.

6. Performance measurements

6.1. Methodology and setup

In this section, we evaluate the performance of Shades. We also compare Shades to Kaleidoscope [16], Local [24], and the caching scheme suggested by the original Kademlia paper [36] (a.k.a. KadCache). For the evaluation, we have used a Java implementation of Shades, Kaleidoscope, KadCache, and Local. We have experimented with several different sizes of networks by running multiple Java VMs (one VM per 80 nodes) on two servers and emulating the users find value requests that are picked from a given, pre calculated workload. We used both synthetic and real life workloads. The real workloads are distributions that were taken from a real YouTube data set [10] and a real Wikipedia data set [40].

In the synthetic distributions, each node in the system periodically picks an item out of 100,000 possible keys according to the specific distribution and issues a find value request for that key. In the YouTube distributions, we used a data set that contains statistics of over 161k newly created videos. These videos were monitored weekly during 21 weeks starting from 16th April, 2008. We used the number of views per week in order to directly generate a distribution that reflects the popularity of each video during that week. As for the Wikipedia trace, it contains an ordered list of requests that were accepted by Wikipedia servers during a period of two months. It is very extensive and contains 10% of the traffic for Wikipedia at that time period. Unfortunately, this trace does not contain client information. Therefore, we simply picked a continuous flow of 5 million requests, cut it into small chunks and randomly but equally assigned them nodes. Each request is then assigned to a key and is searched for during the experiment. This scenario is similar to the case where a load balancer feeds a P2P network with requests.

In all experiments, caches are given a warm-up period in which each node in the system issues 500 find value requests. After the warm-up period, each node in the system issues 500 additional find value requests. Statistics of message send/receive, incoming/outgoing bandwidth and the number of contributing nodes are monitored locally by each node and are collected via HTTP at the end of the experiment. Our experiments where performed on the real system code with the following parameters: bucket size $k = 7$; network sizes: 500, 2500 and 5000 nodes; request distributions: Zipf 0.7, Zipf 0.9. Zipf distributions with similar values were found, e.g., in Web caching and file sharing applications [7,28]. Notice that in the case of 5000 nodes, the experiment includes a total of 5, 000, 000 requests,

Table 2
Hit rate of a single (100 items) cache under different workloads and cache policy settings.

Hit rate (%)			
	LRU	LazyEvict+TinyLFU	Full LFU
Zipf 0.7	0.024	0.095	0.099
Zipf 0.9	0.155	0.283	0.285
YouTube	0.110	0.243	0.245
Wikipedia	0.158	0.322	0.325

half during the warmup period and the other half during the measurement interval.

6.2. Metrics and definitions

We have studied the cache hit rates of Shades and Kaleidoscope, the amount of traffic generated both in terms of message count and overall bandwidth, and the number of contributing nodes.

In considering the cache hit-rate, let us note that in both Shades and Kaleidoscope, the local cache of each node p includes two types of values: items that p has requested by itself and items that have the same color as p and were stored there due to the protocol. We call the collection of items of the first type *self cache* and the collection of the other type of items the *chromatic cache* of p. Hence, when the local cache size is limited, there is contention between the self cache and chromatic cache. Further, in the case of Shades, we distinguish between items found in the *chromatic cache* during the first side step and those found during the second side step. In the measurements, we study both the self caches hit-rate and the chromatic caches hit rate, and in the case of Shades, we also separate between the hit-rate obtained during the first side step vs. the second side-step.

6.3. The choice of a cache policy

We compared the performance of a single cache on our workloads when the cache management policy is LRU, LazyEvict+TinyLFU, and Full LFU. In this test, the cache is tested in clean conditions: it is not part of the system and does not receive requests from other peers. Instead, a single 100 items cache is presented with a stream of requests. We measure the performance of the cache after a short warmup phase. Our results are listed in Table 2. As can be observed, LFU policies are significantly more appealing for our workloads. Moreover, LazyEvict+TinyLFU is able to provide very similar performance to a Full LFU cache, yet with a fraction of the meta-data storage and computation overheads. As mentioned before, a more complete study of TinyLFU (but not LazyEvict) appears in [15].

Let us note that even an unbounded cache cannot obtain a 100% hit rate since the first access to each item is a miss. Further, when the cache is bounded, highly skewed distributions offer a greater potential for hit rates since the set of frequently accessed items is smaller and so it is easier to guess which items are likely to be accessed next. This is also evident in the results shown in Table 2.

Table 3
Effect of the number of colors on the performance of Shades.

Performance and the number of colors				
	Wikipedia		YouTube	
	Shades (50)	Shades (150)	Shades (50)	Shades (150)
Self	0.28	0.26	0.21	0.2
First side step	0.47	0.5	0.59	0.64
Second side step	0.5	0.53	0.65	0.69

6.4. Number of colors

Varying the number of colors has a complex effect. On one hand, increasing the number of colors enhances the observed frequency of items with matching colors more aggressively, thereby increasing their weight in the cache. On the other hand, since the cache size is limited, it comes at the expense of general items, hurting the performance of the self cache.

Hence, the number of colors is a tradeoff parameter. Picking the correct number mainly depends on what the system goals are. In order to explain this tradeoff, we measured the hit rates of the self cache, the first side step and the second side step for different color configurations. This check neglects searches that end due to other reasons within their first few steps.

The results in Table 3 present the different hit rates achieved using 50 and 150 colors. As expected, 50 Colors achieves higher self cache hit rates, but lower chromatic cache hit rates. We feel that Shades offers a more attractive tradeoff with 150 colors than with 50 colors.

This configuration achieves over 50% hit rate within the first two side steps with both the Wikipedia and YouTube workloads. In the latter, it is able to reach 65% hit rate for the first side step and over 70% hit rate after the second side step.

Therefore, as long as the increase in hit rate after the first side step is significant, we suggest increasing the number of colors in order to achieve shorter searches. We focus the rest of our measurements on the 150 colors configuration of Shades. Let us also note that increasing the number of colors has some bandwidth overheads. We address the overall bandwidth overheads of Shades in Section 6.8.

6.5. Self and chromatic caches performance

In this section, we evaluate the hit rates obtained by the chromatic caches of Shades with 150 colors and Kaleidoscope with 17 colors (the recommended Kaleidoscope configuration by the authors of [16]). The results are summarized in Table 4. For a reference, we also point out in the table two extreme values: MC – the maximal possible hit rate for Shades when the chromatic cache spans the entire local cache (self cache is empty), and ML – the maximum achievable hit rate when the entire cache is dedicated to the self cache (chromatic cache is empty).

As can be seen, in both schemes, the hit rate of the chromatic cache is considerably better than the hit rate

Table 4
Cache hit rates for Shades and Kaleidoscope.

Distributed cache performance				
	ML	Kaleidoscope (17)	Shades (150)	MC
Zipf 0.7	0.1	0.16	0.42	0.49
Zipf 0.9	0.29	0.4	0.55	0.75

of the self cache. This is due to the use of colors that increases the likelihood that a searched item will be in the cache. In both schemes, lookup requests have a preference to visit items of the same colors and both schemes attempt to store found values in caches of nodes of the corresponding color. Further, Shades overwhelmingly outperforms Kaleidoscope. The reasons for this are explained below.

Node behavior in Shades is greedy, where each cache stores the most frequent items it observes. Further, the routing protocol of Shades is designed to increase the observed frequency of items with matching color and therefore these items are more likely to appear in caches. When the cache size is limited, however, there is some tension between the self cache hit rate, which enables terminating after zero hops, and the chromatic cache hit rate, which enables terminating after the first (or second) side step.

In general, the best hit rate after the first side step is achieved if the entire cache content is of matching colors since this configuration results in the largest and most effective chromatic cache. However, in this case, since most lookups start at nodes with non-matching colors we have an extremely low probability to terminate locally (only if the searching node is also one of the k-closest nodes to the item's key). Therefore, instead of forcing the local cache to contain only monochromatic items, we let it adjust its content according to TinyLFU's observed frequency. The routing protocol is therefore responsible to augment the observed frequency of items of matching colors. The more colors we use, the more this frequency is augmented and the chromatic cache becomes more effective at the expense of the self cache.

Since Kaleidoscope works best with a smaller number of colors than Shades [16], the chromatic cache in Kaleidoscope is smaller. In addition, as the cache replacement policy in Kaleidoscope is LRU, which is less effective than LazyEvict+TinyLFU used by Shades, its hit rate suffers further.

Overall, Shades achieves very attractive performance. Even in the mildly skewed Zipf 0.7 distribution where a single LRU cache provides less than 3% hit rate, our chromatic cache was able to reach a 42% hit rate after the first side step. The more skewed Zipf 0.9 distribution holds greater caching potential, and indeed Shades achieves a 55% hit rate after the first side step. This means that for both distributions, Shades allows a large portion of the searches to end after a single hop.

Notice that the benefit of Shades' chromatic cache over Kaleidoscope and a dedicated self cache in the Zipf 0.7 distribution is more dramatic than with Zipf 0.9. The reason is that when the distribution is not very skewed, the maximal hit rate of a "usual" cache is limited since it is much harder to predict which items are likely to be accessed next. In Shades, the side steps of the routing protocol and the use of a relatively large number of colors skew the observed frequency of each chromatic cache considerably, enabling the cache to obtain good hit rates.

6.6. Comparison to other caching mechanisms

In this section, we compare Shades to previously suggested caching schemes as well as to a plain Kademlia. We use concurrency of $\alpha = 3$ and measure how many nodes contributed to the lookup resolution. In order to verify statistical significance, the experiments are repeated 3 times, and the number of contributing nodes was averaged for 15k nodes per protocol.

Confidence intervals (with certainty of 0.99) were calculated, but are too small to present in graphs. In particular, the maximal confidence interval for all the data points presented in this section is slightly less than 1% of the average.

As can be observed from Fig. 6, Shades outperforms all previously suggested caching schemes. Moreover, previously suggested caching mechanisms reduce the number of contributing nodes only marginally compared to plain Kademlia. Local mainly benefits from preventing nodes from requesting the same item twice, while KadCache mainly reduces the worst case lookups, which are very rare and therefore does little to reduce the average number. Shades, on the other hand, attempts to hit the cache quickly or not at all. Hence, there is a very small minority of lookups that would end quicker with KadCache. Shades occasionally makes the lookup process longer than in plain Kademlia, as its side steps contact nodes that are not part of the Kademlia lookup path. However, in the typical case of all tested workloads this pays off due to the significant increase in the cache hit rates.

Table 5 presents the median number of contributing nodes values for all the protocols evaluated. Shades reduces this median by as much as 22%–34% compared to the best alternative for every workload.

Unlike median, averages can be manipulated in many ways and are sensitive to edge values. For example, lookups that are resolved at the self cache significantly reduce the average number of contributing nodes without impacting their median. Also, the minority of very long lookups increase the average number of contributing nodes without increasing the median. Our results are presented in Table 6. As can be seen, Shades reduces also the average number of contributing nodes by $\approx$18–23% in comparison to the best alternative of each workload.

We expect Shades' advantage to become more dominant with larger networks. Since the lookup paths of Kademlia grow longer with the network size, the impact of finishing a large portion of the searches within the first two hops becomes greater in large networks.

6.7. Unbounded cache size experiment

The various schemes we tested utilize different cache replacement schemes. In particular, KadCache, Local and Kaleidoscope use an LRU cache while Shades uses our

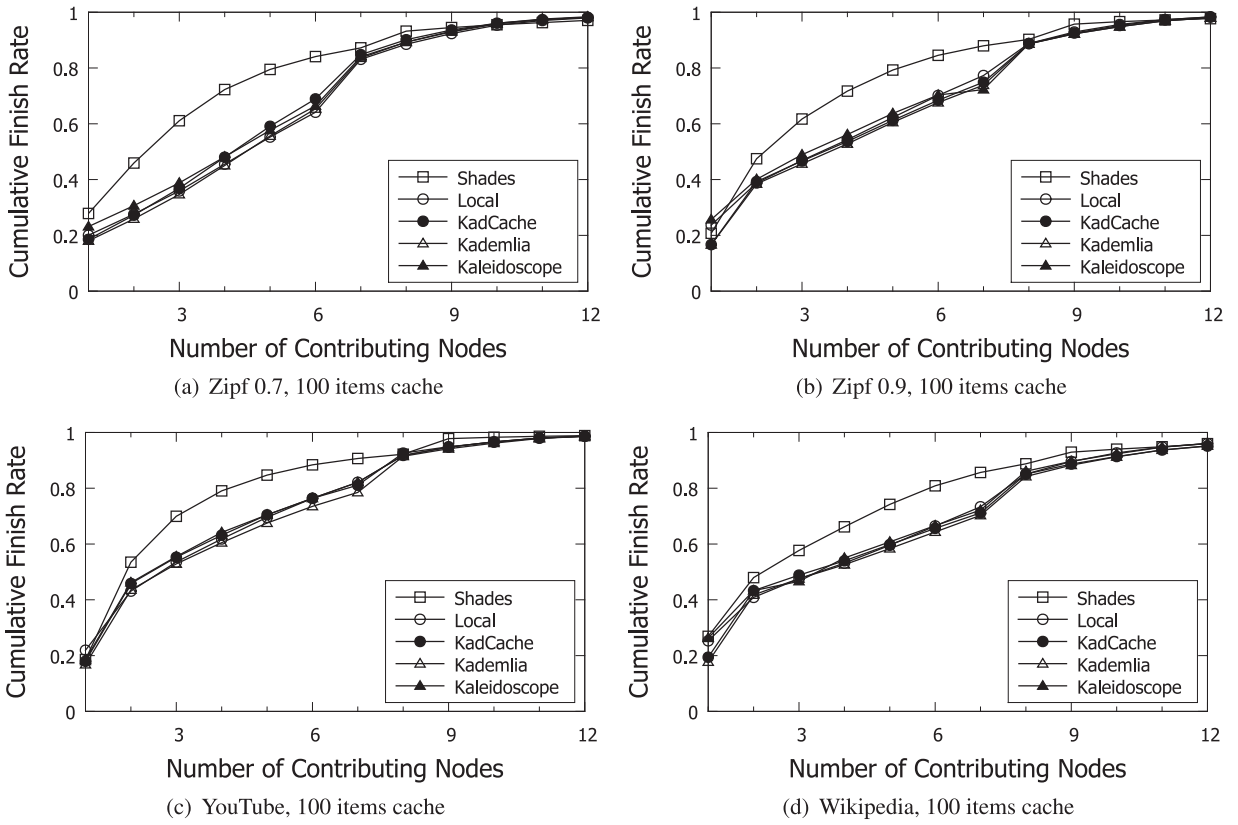


Fig. 6. Number of contributing nodes required to perform a lookup.

Table 5
Median number of contributing nodes during the measurements (the last column is the ratio between Shades and its best alternative).

Median number of contributing nodes						
	Kademlia	Local	KadCache	Kaleidoscope	Shades	Shades/Best
Zipf 0.7	5.47	5.29	5.18	5.1	3.27	0.64
Zipf 0.9	3.76	3.42	3.76	3.15	2.18	0.69
YouTube	2.69	2.66	2.44	2.64	1.9	0.78
Wikipedia	3.48	3.44	3.23	3.2	2.21	0.69

Table 6
Average number of contributing nodes during the measurements (the last column is the ratio between Shades and its best alternative).

Average number of contributing nodes						
	Kademlia	Local	KadCache	Kaleidoscope	Shades	Shades/Best
Zipf 0.7	5.34	5.29	5.16	5.12	4.08	0.79
Zipf 0.9	4.01	3.92	4.01	4.20	3.03	0.77
YouTube	3.72	3.41	3.40	3.40	2.74	0.81
Wikipedia	4.32	4.06	4.14	4.15	3.31	0.82

own LazyEvict+TinyLFU. Therefore, it is important to understand what gives Shades the edge over other strategies. In order to level the playing field regarding the cache policy, we experiment with unlimited caches, which render the eviction policy immaterial, using exactly the same workloads as our other experiments. The only bound for the cache size in this experiment is the length of the experiment that remains the same.

The results are illustrated in Fig. 7. As can be observed, both Local and KadCache perform worse than an unbounded Shades cache. Moreover, they are even worse than a 100 items Shades cache for all tested workloads.

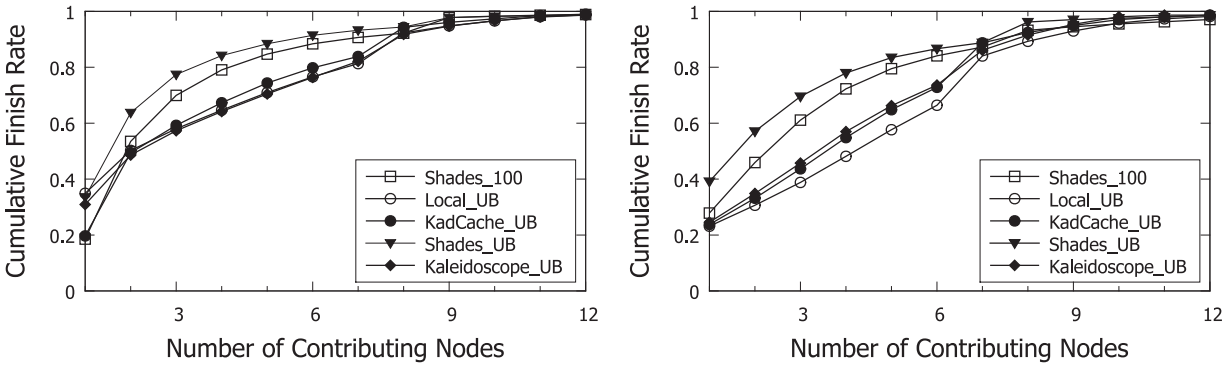


Fig. 7. Competitors with unbounded caches vs. Shades with 100 items cache and an unbounded cache.

Table 7
Average number of handled messages during the measurements (the last column is the ratio between Shades and its best alternative).

Average number of handled messages (1000's)						
	Kademlia	Local	KadCache	Kaleidoscope	Shades	Shades/Best
Zipf 0.7	7.1	7.1	7.2	5.5	6.0	1.09
Zipf 0.9	5.6	5.3	5.9	4.8	4.7	0.98
YouTube	5.2	4.9	5.1	4.3	4.3	1
Wikipedia	6	5.6	6.1	5.2	5.2	1

Table 8
Average bandwidth costs during the measurements (the last column is the ratio between Shades and its best alternative).

Average incoming traffic (MB)						
	Kademlia	Local	KadCache	Kaleidoscope	Shades	Shades/Best
Zipf 0.7	12.00	11.59	11.99	9.78	10.93	1.12
Zipf 0.9	9.16	8.55	9.9	8.61	8.49	0.99
YouTube	8.52	7.90	8.52	7.87	7.66	0.97
Wikipedia	9.84	9.06	10.14	8.5	9.24	1.09

This result implies that a major contribution of Shades indeed comes from its routing protocol.

Note that TinyLFU is inherent in Shades routing protocol, as detailed in Section 4.2. Hence, running Shades with LRU would simply add management overhead and therefore does not make sense.

6.8. Communication overheads

The message overheads of Shades and the other protocols are quantified in Table 7. As can be observed, Shades sends a comparable number of messages to the best alternative for every workload. Yet, as Shades modifies the original Kademlia messages to include its own information inside them, each Shades message is slightly bigger. Table 8 quantifies the overall average bandwidth received by each node during our measurements. This bandwidth includes all the data sent by the protocols.

As can be observed, Shades is always comparable to the best alternative for that workload. Thus, although message count and communication bandwidth are not part of our design goals, Shades encounters no significant overheads in these metrics.

6.9. Congestion control

The effect of our explicit congestion notification mechanism is evaluated in Fig. 8. This figure is a sorted histogram of the number of messages handled by each node during both the YouTube and the Zipf 0.7 measurements. As can be observed, Shades offers a significantly better load distribution when equipped with this simple congestion control mechanism. This mechanism also had similar effects on the load distributions of all other protocols. The downside of this mechanism is a slight decrease in the average cache hit rate, as lookups are routed more evenly in the network. However, the total impact on hit rate is almost unnoticed.

Yet, even when all other caching suggestions are equipped with the explicit congestion notification mechanism, Shades offered a significantly more balanced load distribution. The effect is quantified by Table 9 that compares the average number of messages handled by the most congested 50 nodes in the network (1% busiest nodes). As can be observed, for each workload, Shades improves the load placed on these nodes by 22–43%. We credit the improvement to Shades' routing technique since

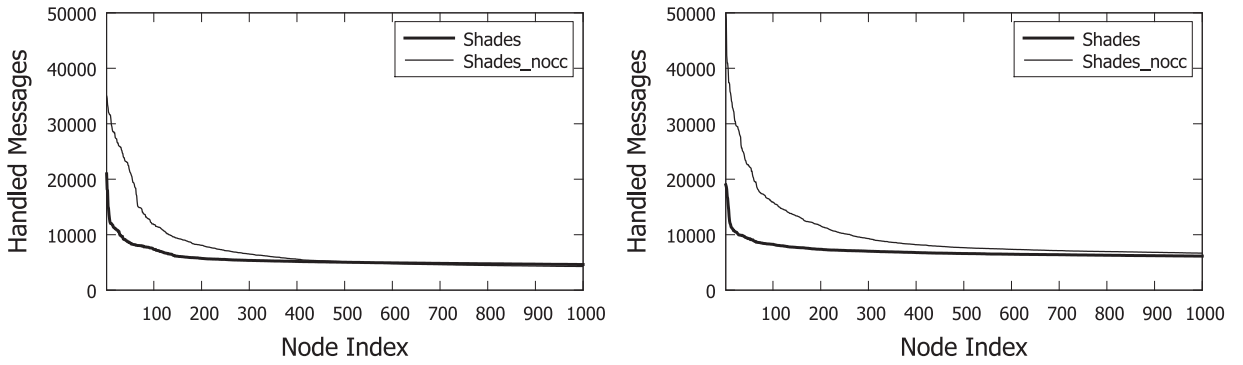


Fig. 8. Load distribution across the 1000 most congested nodes (out of 5000 overall) for Shades (150 colors) with and without congestion control mechanism.

Table 9
Load placed upon the most congested nodes (the last column is the ratio between Shades and its best alternative).

Messages handled by 1% most congested nodes						
	Kademlia	Local	KadCache	Kaleidoscope	Shades	Shades/Best
Zipf 0.7	26.2	23.9	22.7	20.05	11.45	0.57
Zipf 0.9	21.4	18.6	16.7	17	13.00	0.78
YouTube	22.4	18.2	21.1	17.9	13.3	0.74
Wikipedia	26.6	17.6	19.9	17	13.3	0.78

all other routing protocols are also equipped with the same congestion control mechanism.

7. Discussion

We have presented Shades, a combined routing/caching scheme that augments Kademlia, yielding a significant reduction in the number of contributing nodes. Through simulations that are based on artificial Zipf-like distributed workloads as well as real traces from YouTube and Wikipedia, we have found that Shades reduces the median number of nodes contributing to each lookup by 22–36% compared to the best of breed among the other schemes in the workloads tested and a 30–40% reduction compared to plain Kademlia. Shades obtains a load reduction on the busiest nodes (hot-spots) of 22–43% with respect to the best scheme and 40–56% compared to plain Kademlia. With reported latencies of 5.8–7.6 s for tuned Kademlia based systems such as [32,37], our improvements can have a significant impact on the user experience of these systems.

Shades also generated fewer messages than KadCache and Local, and a similar bandwidth consumption as the best of breed among them. In some workloads, Kaleidoscope offers slightly lower message and bandwidth costs, but the differences are small.

Another feature of Shades is that with a small 100 items cache it performs better than any of the other caching schemes even when they are equipped with an unbounded cache. Shades is an open source project [2], implemented as an extension to OpenKad [1].

When using caching, there is always the question of keeping the cache content consistent. There are many applications in which data is immutable, in which case the problem does not exist. In particular, in such systems explicit versioning is often used instead of updates (e.g., http://www.saphana.com/). In other cases, using periodic revalidation against the main copy or deleting items from the cache after a TTL is enough to ensure timely eventual consistency [42].

References

[1] OpenKad, http://code.google.com/p/openkad/.
[2] Shades source code, https://code.google.com/p/shades/.
[3] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, S. Ron, Practical locality-awareness for large scale information sharing, in: Proceedings of the 4th Annual International Workshop on Peer-To-Peer Systems (IPTPS 2005), 2005.
[4] S. Androutsellis-Theotokis, D. Spinellis, A survey of P2P content distribution technologies, ACM Comput. Surv. 36 (2004) 335–371.
[5] M. Arnedo, M. Pilar Villamil, R. Villanueva, H. Castro, L. d'Orazio, A catalog-based caching strategy for structured p2p systems, in: Proceedings of the 3rd International Conference on Data Management in Grid and Peer-to-Peer Systems (Globe 2010), 2010, pp. 50–61.
[6] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426, doi:10.1145/362686.362692.
[7] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and zipf-like distributions: Evidence and implications, in: Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1999), 1999, pp. 126–134.
[8] D. Carra, M. Steiner, P. Michiardi, Adaptive load balancing in kad, in: Proceedings of the 11th IEEE International Conference on P2P Computing (P2P 2011), 2011, pp. 92–101, doi:10.1109/P2P.2011.6038666.
[9] M. Castro, P. Druschel, Y.C. Hu, A. Rowstron, Exploiting network proximity in distributed hash tables, in: International Workshop on Future Directions in Distributed Computing (FuDiCo 2002), 2002, pp. 52–55.
[10] X. Cheng, C. Dale, J. Liu, Statistics and social network of youtube videos, in: Proceedings of the 16th International Workshop on Quality of Service (IWQoS 2008)., 2008, pp. 229–238, doi:10.1109/IWQOS.2008.32.

[11] S. Cohen, Y. Matias, Spectral bloom filters, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD 2003), 2003, pp. 241–252.
[12] F. Dabek, E. Brunskill, M. Kaashoek, D. Karger, R. Morris, I. Stoica, H. Balakrishnan, Building peer-to-peer systems with chord, a distributed lookup service, in: Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS 2001), 2001, pp. 81–86.
[13] S. Deb, P. Linga, R. Rastogi, A. Srinivasan, Accelerating lookups in p2p systems using peer caching, in: Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE 2008), 2008, pp. 1003–1012.
[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store, in: Proceedings of 21th ACM SIGOPS Symposium On Operating Systems Principles (SIGOPS 2007), 2007, pp. 205–220, doi:10.1145/1323293.1294281.
[15] G. Einziger, R. Friedman, Tinylfu: A highly efficient cache admission policy, in: Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2014), 2014.
[16] G. Einziger, R. Friedman, E. Kibbar, Kaleidoscope: Adding colors to kademlia, in: Proceedings of the 13th IEEE International Conference on P2P Computing (P2P 2013), 2013.
[17] C. Estan, G. Varghese, New directions in traffic measurement and accounting, ACM Trans. Comput. Syst. (TOCS 2003) 21 (3) (2003) 270–313.
[18] B. Fan, D.G. Andersen, M. Kaminsky, Memc3: compact and concurrent memcache with dumber caching and smarter hashing, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI 2013), 2013, pp. 371–384.
[19] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. (TON) 8 (3) (2000) 281–293.
[20] S. Floyd, Tcp and explicit congestion notification, Comput. Commun. Rev. (ACM SIGCOM 1994) 24 (5) (1994), doi:10.1145/205511.205512.
[21] M.J. Freedman, Experiences with coralcdn: A five-year operational view, in: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation(NSDI 2010), 2010, p. 7.
[22] M.J. Freedman, E. Freudenthal, D. Mazires, Democratizing content publication with coral, in: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI 2004), 2004, p. 18.
[23] C. Groß, D. Stingl, B. Richerzhagen, A. Hemel, R. Steinmetz, D. Hausheer, Geodemlia: A robust p2p overlay supporting location-based search, in: Proceedings of the 12th IEEE International Conference on Peer-to-Peer Computing (P2P 2012), 2012, pp. 25–36.
[24] L. Guangmin, An Improved Kademlia Routing Algorithm for P2P Network, in: International Conference on New Trends in Information and Service Science (NISS 2009), 2009, pp. 63–66.
[25] A. Gupta, B. Liskov, R. Rodrigues, One hop lookups for peer-to-peer overlays, in: Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS 2009), 2009.
[26] I. Gupta, K. Birman, P. Linga, A. Demers, R. van Renesse, Kelips: Building an efficient and stable p2p dht through increased memory and background overhead, in: Proceedings of the 2nd International Workshop on P2P Systems (IPTPS 2003), 2003.
[27] B. Heep, R/kademlia: Recursive and topology-aware overlay routing, in: Australasian Telecommunication Networks and Applications Conference (ATNAC 2010), 2010, pp. 102–107, doi:10.1109/ATNAC.2010.5680244.
[28] A. Iamnitchi, M. Ripeanu, I.T. Foster, Small-world file-sharing communities, in: Proceedings of the 23th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004), 2004, pp. 952–963.
[29] S. Jain, R. Mahajan, D. Wetherall, A study of the performance potential of dht-based overlays, in: In Proceedings of the 4th Usenix Symposium on Internet Technologies and Systems (USITS 2003), 2003.
[30] S. Kaune, T. Lauinger, A. Kovacevic, K. Pussep, Embracing the peer next door: Proximity in kademlia, in: Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing (P2P 2008), 2008, pp. 343–350, doi:10.1109/P2P.2008.36.
[31] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, Zht: A light-weight reliable persistent dynamic scalable zero-hop dht, in: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS 2013), 2013, pp. 775–787, doi:10.1109/IPDPS.2013.110.
[32] B. Liu, T. Wei, J. Zhang, J. Li, W. Zou, M. Zhou, Revisiting why kad lookup fails, in: Proceedings of the 12th IEEE International Conference on Peer-to-Peer Computing (P2P 2012), 2012, pp. 37–42, doi:10.1109/P2P.2012.6335808.
[33] N. Lopes, C. Baquero, Taming hot-spots in dht inverted indexes, in: Proceedings of the 11th International Workshop on Large-Scale and Distributed Systems (LSDS 1007).
[34] E. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, A survey and comparison of P2P overlay network schemes, IEEE Commun. Surv. Tutor. 7 (2) (2005) 72–93.
[35] Z. Lv, T. Yin, Y. Han, Y. Chen, G. Chen, Webvr – web virtual reality engine based on p2p network, J. Netw. 6 (7) (2011).
[36] P. Maymounkov, D. Mazières, Kademlia: A P2P Information System Based on the XOR Metric, in: Proceedings of the 1st International Workshop on P2P Systems (IPTPS 2002), 2002, pp. 53–65.
[37] M. Steiner, D. Carra, E.W. Biersack, Faster content access in kad, in: Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing (P2P 2008), 2008, pp. 195–204, doi:10.1109/P2P.2008.28.
[38] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOM 2001), 2001, pp. 149–160, doi:10.1145/383059.383071.
[39] D. Stutzbach, R. Rejaie, Improving lookup performance over a widely-deployed dht, in: Proceedings of the 25th IEEE International Conference on Computer Communications. (INFOCOM 2006), 2006, pp. 1–12, doi:10.1109/INFOCOM.2006.329.
[40] G. Urdaneta, G. Pierre, M. van Steen, Wikipedia workload analysis for decentralized hosting, Comput. Netw. 53 (11) (2009) 1830–1845.
[41] M. Varvello, E. Biersack, C. Diot, A networked virtual environment over kad, in: Proceedings of the ACM CoNEXT Conference (CoNEXT 2007), 2007.
[42] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (2009) 40–44.
[43] J.M. Wozniak, B. Jacobs, R. Latham, S. Lang, S.W. Son, R.B. Ross, C-mpi: A dht implementation for grid and hpc environments, 2010. Preprint ANL/MCS-P1746-0410.

Gil Einziger is a postdoctoral researcher at Politecnico di Torino, Turin Italy. He received his Ph.D degree in computer science from the computer science department of the Technion, Haifa, Israel in 2015 and is now a post doctoral researcher at the Politecnico di Torino, Turin, Italy. His research interests include P2P systems, network monitoring and streaming algorithms. In the past Gil has also worked in Intel and Digital feedback technologies.

Roy Friedman Is with the Computer Science Department at the Technion. His research interests include Distributed Systems with emphasis on Mobile Computing, Mobile Ad-Hoc Networks, Fault-Tolerance and High Availability, and Peer-to-Peer computing. Formerly, Roy Friedman was an academic specialist at INRIA (France) and a researcher at Cornell University (USA). He is a founder of PolyServe Inc. (acquired by HP) and holds a Ph.D. and a B.Sc. from the Technion.

Yoav Kantor was born in Israel in 1983. He received a B.Sc. degree and an M.Sc. degree in Computer Science from the Technion - Israel Institute of Technology, in 2011 and 2014 respectively. He has been working and conducting research in the field of distributed systems and computer networks, focusing on distributed storage systems, distributed cache, and ad-hoc networks. He is currently Research Scientist at IBM Research.