Shades: Expediting Kademlia's Lookup Process*

Gil Einziger, Roy Friedman, and Yoav Kantor

Computer Science Department, Technion, Haifa 32000, Israel
{gilga,roy,ykantor}@cs.technion.ac.il

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 has been published, e.g., [4,19]. 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 overlay. During the last few years, Kademlia has become one of the most widely used DHTs in practice [20,22]. 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 [25].

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 [18,21]. For example, one of the lessons of the CoralCDN project [10], a successful DHT based content delivery network, is that DHT lookup latency was a performance bottleneck for their system.

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 [9].

To tackle this problem, we introduce a novel caching and augmented routing mechanism for Kademlia called Shades (the entire code of Shades is available as open source at [2]). That is, each node maintains a small local cache that is

* This work is partially supported by the Technion HPI Research School.

© Springer International Publishing Switzerland 2014

managed using an effective cache filtering mechanism called *TinyLFU*. TinyLFU maintains a compressed approximate statistics of all items encountered and uses this as an admission filter that only admits popular items into the cache and is able to do so in a very time and space efficient manner, as reported in [8]. Further, Shades augments Kademlia's routing decisions using a secondary hashing technique that we call *colors*. As described later in this paper, colors are used to help caches specialize in items of their own color, thereby increasing the skewing in the observed access distribution of the requests they encounter, resulting in much higher hit rates. In particular, a limited number of lookup requests are issued to nodes whose id hashes to the same color as the key of the item being searched rather than according to the usual Kademlia lookup process. In addition, the hints from TinyLFU's statistics are used to limit the number of such deviations from the normal lookup process and only apply them when there is statistical evidence that they are likely to help. Finally, we employ an *overload protection* mechanism to prevent Kademlia nodes from becoming overwhelmed with requests[1].

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 [20], the local cache suggested in [12] – a.k.a. *Local* and *Kaleidoscope* [9]. 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 Shades (and Kademlia) in Section 2. Section 3 survey an additional related work. Performance evaluation is presented in Section 4. Finally, we conclude with a discussion in Section 5.

2 Shades

As indicated before, 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 2.1, the routing scheme is presented in Section 2.2, and the overload protection is explained in Section 2.3.

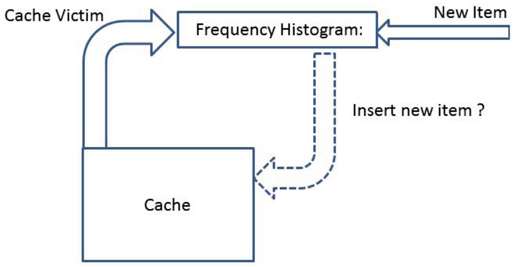
2.1 Caching Mechanism

With Shades, each node in the system maintains a small local cache in addition to its Kademlia storage. When a node receives a lookup request, it can either

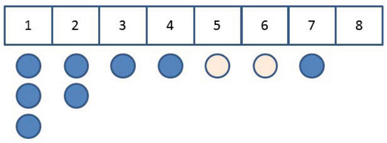
[1] This latter optimization does not improve the lookup hop-count, but rather the overall latency by avoiding routing to nodes that are overloaded.

return the k-closest nodes, return a cached result or return the stored value. The difference between Kademlia storage and the cache is that the former has to store all items that the node is assigned to according to the Kademlia DHT algorithm. On the other hand, the goal of the cache is to boost the performance of the system by storing selective items. In particular, in order to keep the cache size small and since real world workloads tend to exhibit access locality, intuitively the cache should include the most frequently requested items.

For the cache management, we have chosen to employ the cache architecture of TinyLFU [8]. In TinyLFU, there is a separation between cache eviction policies and admission policies. TinyLFU maintains an approximated statistics over all recently encountered items such that a new item will replace the cache victim only if it is more frequent than the cache victim as illustrated in Figure 1(a). Since the statistics are kept over a large collection of past requests and can potentially be very big, TinyLFU only maintains an approximation of this statistics. To keep the statistics fresh, TinyLFU perform a periodic *Reset* operation, this operation halves all counters. As reported in [8], the memory overhead associated with TinyLFU is comparable to a memory pointer per cache line. Since TinyLFU can work with any eviction policy, we complement TinyLFU, with a *Lazy LFU* eviction policy. This policy attempts to find the least frequently used item in the cache, however does so lazily, performing a single search step per cache lookup, resulting in $O(1)$ query complexity, and hit rate similar to a true LFU cache.



(a) TinyLFU general architecture: a new item is only admitted if it is recently more popular than the cache victim.



(b) Shades Palette

Fig. 1. An illustration of TinyLFU and Shades Palette

2.2 Routing

As mentioned before, Shades augments the standard Kademlia routing scheme by utilizing a secondary key called *color* in order to partition cache content between nodes and create a distributed large cache out of many small individual caches. Unlike the Kademlia key that comes from a large domain to prevent collisions, the color domain is small and collisions are desirable.

During the parallel iterative lookup process, Shades may issue cache lookup requests to nodes that have the same color as the requested key even if these nodes do not advance in Kademlia's XOR distance metric [20].[2] For this reason, we call such deviations a *side step*. 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 this section.

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 frequent then the cached items, Shades ensures that the cached result is shared with a node that is likely to admit it. In the rest of this section, we first describe an auxiliary data structure used by the routing mechanism of Shades and then provide the details of the protocol.

Palette. Since going over all the k-buckets in order to find a matching color candidate can be time consuming, each node p maintains a mapping between colors and the nodes matching these colors that p is award 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 Figure 1(b). 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.

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

[2] The distance metric used by Kademlia to decide on hashed ids proximity is XOR.

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. We call these cache lookups side steps since they are not necessarily advancing 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, which 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 2.1.
– Is the item popular? I.e., is this item likely to be admitted to other caches.

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 protocol to usually perform the first side step right in the start of a lookup, which is important since the benefit of hitting a cache early is far greater than hitting it later.

2.3 Congestion Control

When we started experimenting with Kademlia in general and Shades in particular, we encountered a severe congestion problem when running test cases with many nodes (this can be seen in the result in Section 4.5 below). For this reason, we implemented a simple congestion control mechanism .

In that mechanism every message is attached an additional bit that is set if the sender's incoming message queue was more then 75% full when the message was sent. Once a node receives a message with a set congestion bit, it marks the sending node as a candidate for replacement. That is the receiver encounter a possibility to replace the congested, it will do so without sending a ping message. The result of this mechanism is that congested nodes reduce their representation in routing tables and therefore receive less incoming traffic.

3 Related Work

Several works have investigated how to use caching to reduce the lookup cost in DHTs. For example, in [12] 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 lookup request, it first checks its Fast Table to see if it contains cached results for it. This approach was shown in [9,12] 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 [20]. In this suggestion, every time a node performs a lookup 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 [9] for its message cost and (lack) of load balance capabilities. In this paper, we extend that evaluation of KadCache to cover its latency aspects. 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 [9]. 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 than Shades. 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 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.

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	TinyLFU+LazyLFU
Share policy	Always	Only if needed

Other methods to reduce Kademlia's lookup latency includes careful parameter configuration [22], techniques to fill k-buckets with nodes of geographical proximity [16], a new metric based on geographical distance [11] and a recursive lookup scheme [15]. 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.

Other DHT's like OneHop [13], Kelips [14] 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 include O(1) lookups include, e.g., Dynamo [7] and ZHT [17]. Both systems target high performance data centers. Given that a variant of Kademlia was also suggested for this context [25], Shades can be adopted to that domain as well.

4 Performance Measurements

4.1 Methodology and Setup

In this section, we evaluate the performance of Shades. We also compare Shades to Kaleidoscope [9], Local [12], and the caching scheme suggested by the original Kademlia paper [20] (a.k.a. KadCache). For the evaluation, we 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 lookup 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 [6] and a real Wikipedia data set [23].

In the synthetic distributions, each node in the system periodically picks an item out of 1,000,000 possible keys according to the specific distribution and issues a lookup 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.

In all experiments, caches are given a warm-up period in which each node in the system issues 500 lookup requests. After the warm-up period, each node in the system issues 500 additional lookup requests. Statistics of message send/receive, incoming/outgoing bandwidth and the latency 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, $2,500$ and $5,000$ 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 [5]. Notice that in the case of $5,000$ nodes, the experiment includes a total of $5,000,000$ requests, half during the warmup period and the other half during the measurement interval.

4.2 Metrics and Definitions

Since the wall-clock latency depends on a large number of factors and is therefore very noisy, we have decided to focus on measuring the number of *contributing nodes* for each lookup, i.e., the number of nodes whose replies were utilized while performing the lookup, instead of wall-clock latency. We note that this number may be different from the number of contacted nodes, e.g., if three parallel lookups are sent and the first reply returns the value, then the number of contributing nodes is 2 (the initiator and the node that returned the cached result), even though 3 nodes were contacted. Since Kademlia works with concurrent iterative lookup, this is not exactly the latency in hops. Yet, since our experiments were conducted with $\alpha = 3$, dividing the number of contributing nodes by α (3) gives a relatively good estimation to the number of hops used in the lookup process. We have also studied the cache hit rates as well as the amount of traffic generated both in terms of message count and overall bandwidth.

4.3 Number of Colors

Varying the number of colors has a complex effect. On the one hand, increasing the number of colors enhances the observed frequency of correctly colored items

Table 2. 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)
Local	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

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 local 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 local 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 2 present the different hit rates achieved using 50 and 150 colors. As expected, 50 Colors achieves higher local 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 Wikipedia and YouTube workloads. In the latter, it reaches 65% hit rate for the first side step and over 70% hit rate after the second side step.

Hence, 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 lower latency. The rest of our measurements focus on the 150 colors configuration of Shades.

4.4 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.

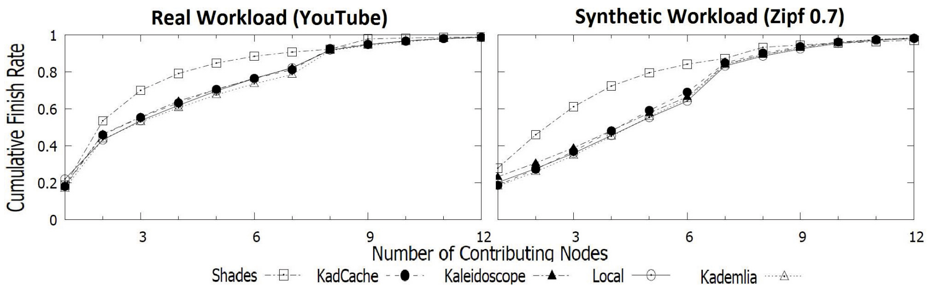


Fig. 2. Number of contributing nodes required to perform a lookup

To get a better feel for the latency improvement of Shades, we exhibit the average and median lookup latency measured by the number of contributing nodes. The median represents how many nodes are required on average to resolve half of the lookups.

Table 3 presents the median latency values for all the protocols evaluated. Shades reduces the median latency by as much as 22% – 34% compared to the best alternative for every workload.

Unlike median, average latency can be manipulated in many ways and is sensitive to edge values. For example, lookups that are resolved at the local cache significantly reduce the average latency without impacting the median latency. Also the minority of very long lookups increase the average latency without increasing the median latency. Our results are presented in Table 3.As can be seen, Shades improves also the average latency by $\approx 18 - 23\%$ in comparison to the best alternative of each workload.

Table 3. Average and median latency during the measurements

Average And Median Latency (Contributing Nodes)												
	Kademlia		Local		KadCache		Kaleidoscope		Shades		Shades/Best	
Metric	A	M	A	M	A	M	A	M	A	M	A	M
Zipf 0.7	5.34	5.47	5.29	5.29	5.16	5.18	5.12	5.1	4.08	3.27	0.79	0.64
Zipf 0.9	4.01	3.76	3.92	3.42	4.01	3.76	4.20	3.15	3.03	2.18	0.77	0.69
YouTube	3.72	2.69	3.41	2.66	3.40	2.44	3.40	2.64	2.74	1.9	0.81	0.78
Wikipedia	4.32	3.48	4.06	3.44	4.14	3.23	4.15	3.2	3.31	2.21	0.82	0.69

We expect Shades' latency 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.

4.5 Load Distribution

Table 4 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%. Since all routing protocols are equipped with the same congestion control mechanism, we credit the improvement to our routing technique. Intuitively, Shades sends lookups in two different directions, distributing the load more evenly in the system.

Table 4. Load placed upon the most congested nodes

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

5 Discussion

We have presented Shades, a combined caching/routing scheme that augments Kademlia, yielding a significant improvement in latency. 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 seconds for tuned Kademlia based systems such as [18,21], 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 than Shades, but the differences are small.

Another interesting aspect of Shades is that its latency with a small cache of 100 items is better than any of the other caching schemes we have compared against 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 [24].

References

1. OpenKad, `http://code.google.com/p/openkad/`
2. Shades source code, `https://code.google.com/p/shades/`
3. Abraham, I., Badola, A., Bickson, D., Malkhi, D., Maloo, S., Ron, S.: Practical locality-awareness for large scale information sharing. In: van Renesse, R., Castro, M. (eds.) IPTPS 2005. LNCS, vol. 3640, pp. 173–181. Springer, Heidelberg (2005)
4. Androutsellis-Theotokis, S., Spinellis, D.: A Survey of P2P Content Distribution Technologies. ACM Computing Survey 36, 335–371 (2004)
5. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: INFOCOM, pp. 126–134 (1999)
6. Cheng, X., Dale, C., Liu, J.: Statistics and social network of youtube videos. In: 16th Int. Workshop on Quality of Service, IWQoS 2008, pp. 229–238 (June 2008)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41(6), 205–220 (2007)
8. Einziger, G., Friedman, R.: Tinylfu: A highly efficient cache admission policy. In: 22nd Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP), pp. 146–153 (February 2014)

9. Einziger, G., Friedman, R., Kibbar, E.: Kaleidoscope: Adding colors to kademlia. In: Proc. of the 13th IEEE Int. Conf. on P2P Computing (September 2013)
10. Freedman, M.J., Freudenthal, E., Mazières, D.: Democratizing content publication with coral. In: Symposium on Networked Systems Design and Implementation, NSDI 2004, pp. 18–18. USENIX Association, Berkeley (2004)
11. Groß, C., Stingl, D., Richerzhagen, B., Hemel, A., Steinmetz, R., Hausheer, D.: Geodemlia: A robust p2p overlay supporting location-based search. In: Proc. of the 12th IEEE Int. Conf. on P2P Computing. IEEE (September 2012)
12. Guangmin, L.: An Improved Kademlia Routing Algorithm for P2P Network. In: Int. Conf. on New Trends in Information and Service Science, pp. 63–66 (2009)
13. Gupta, A., Liskov, B., Rodrigues, R.: One hop lookups for peer-to-peer overlays. In: Proc. of the 9th Conf. on Hot Topics in Operating Systems, HOTOS 2003. USENIX Association, Berkeley (2003)
14. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 160–169. Springer, Heidelberg (2003)
15. Heep, B.: R/kademlia: Recursive and topology-aware overlay routing. In: 2010 Australasian Telecommunication Networks and Applications Conf (ATNAC), 31 October- November 3, pp. 102–107 (2010)
16. Kaune, S., Lauinger, T., Kovacevic, A., Pussep, K.: Embracing the peer next door: Proximity in kademlia. In: Eighth Int. Conf. on P2P Computing, P2P 2008, pp. 343–350 (September 2008)
17. Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z., Raicu, I.: Zht: A light-weight reliable persistent dynamic scalable zero-hop dht. In: Parallel & Distributed Processing Symposium, IPDPS (2013)
18. Liu, B., Wei, T., Zhang, J., Li, J., Zou, W., Zhou, M.: Revisiting why kad lookup fails. In: Proc. of the 12th Int. Conf. on P2P Computing, pp. 37–42. IEEE (2012)
19. Lua, E., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A Survey and Comparison of P2P Overlay Network Schemes. IEEE Communications Surveys Tutorials 7(2), 72–93 (2005)
20. Maymounkov, P., Mazières, D.: Kademlia: A P2P Information System Based on the XOR Metric. In: Proc. of the 1st Int. Workshop on P2P Systems (IPTPS), pp. 53–65 (2002)
21. Steiner, M., Carra, D., Biersack, E.W.: Faster content access in kad. In: Proc. of the 8th Int. Conf. on P2P Computing, pp. 195–204. IEEE Computer Society, Washington, DC (2008)
22. Stutzbach, D., Rejaie, R.: Improving lookup performance over a widely-deployed dht. In: INFOCOM 2006. 25th IEEE Int. Conf. on Computer Communications. Proc., pp. 1–12 (2006)
23. Urdaneta, G., Pierre, G., van Steen, M.: Wikipedia workload analysis for decentralized hosting. Elsevier Computer Networks 53(11), 1830–1845 (2009)
24. Vogels, W.: Eventually consistent. Communications of the ACM 52(1), 40–44 (2009)
25. Wozniak, J.M., Jacobs, B., Latham, R., Lang, S., Son, S.W., Ross, R.B.: C-mpi: A dht implementation for grid and hpc environments. In: Preprint ANL/MCS-P1746-0410, 04/2010 (2010)