Cooperative client caching strategies for social and web applications

Stavros Nikolaou
Cornell University
snikolaou@cs.cornell.edu

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

Nicolas Schiper
Cornell University
nschiper@cs.cornell.edu

Abstract

We evaluate the efficiency and cost of different placement strategies for a distributed cache implemented on the clients of an online social network or web service. In our system model, the service maintains a directory for content that tracks the location of objects. The service informs requesting clients of these locations and the clients will cache, serve, and push content according to directives provided by the service. We show that we can improve the individual cache hit ratio by leveraging relationships between clients (e.g., social links). The network load imposed on clients is practical.

1 Introduction

Today's web services, including social networks, photo-sharing web-sites, and video on demand systems, provide vast amounts of content. As the web service becomes more popular, the cost of content distribution increases and the service has to rely either upon content delivery networks (CDNs) or build its own delivery system. This is costly to deploy and manage, and previous work [16, 6] has focused on clients helping with content distribution by acting as caches and serving content to each other in a peer-to-peer (P2P) fashion. This can alleviate load on the service as well as protect it from flash crowds [12]. At the same time, cooperative caching can improve network resource utilization by distributing bandwidth usage to many client-to-client connections. Recent systems [7, 16] have shown that such approaches are feasible, effective, and in Maygh's [16] case, they can be non-intrusively deployed in web browsers using technologies like RTMFP and WebRTC.

This work considers a system model similar to that of Maygh. Clients issue requests for content and cache it for future use or for serving it to other clients. The service keeps track of the content and its locations, and replies to requests by providing either the content or a location where it can be found. Caches at clients have a limited capacity.

In such a cooperative setting, strategically placing cached content can substantially affect performance and overhead [10]. In order not to overload clients with download requests from other clients, popular objects need to be cached widely, but if too many copies are made, the capacity of the cooperative cache is needlessly wasted. The caching system should adapt to the dynamic behavior of clients and changes in content popularity.

In this paper, we investigate how cache placement affects hit ratio and the load perceived by the service and the clients. We propose a novel cache placement strategy that leverages relationships between clients. We compare this algorithm to three other placement strategies. The evaluation is performed using simulations on synthetically generated workloads that match certain characteristics of Online Social Networks (OSNs) [3, 11]. Results of this evaluation suggest that, by employing knowledge of relationships between clients (such as the social graph in the case of an OSN), we can improve local cache hit ratios with minor toll on the hit ratio of the collective cache. The overhead on clients is non-negligeable but still moderate.

2 Related Work

Cooperative caching has been studied extensively in the setting of web caching [15, 14]. Example systems include Shark [2] and Backslash [12] that build cooperative P2P caches on proxy servers.

Squirrel [6], another P2P web caching mechanism that uses a DHT for achieving scalability, self-organization, and churn tolerance, is intended for geographically collocated clients (e.g., at the same company). This can be limiting for certain web services such as OSNs.

In [9], several placement algorithms for CDNs are proposed and evaluated. These schemes use workload infor-

mation such as client latency and request rates. However, none of them employ social information to improve placement decisions. Closest to our proposed placement approach is S-Clone [13], which collocates replicas of data of neighboring clients with respect to the social graph of an OSN. Different from our work, data is replicated on servers and not on clients. Their approach does not take workload information into account.

3 System Model

3.1 The Service, Objects, and Clients

The service is responsible for two main functions: serving objects to clients and issuing *caching directives* that specify which clients should cache these objects. In our model, objects are immutable and each object has an owner, which is one of the clients. The owner does not necessarily cache the object. Objects are identified by *keys*. The service keeps track of which clients are currently online. The service uses a key-value store to maintain for each object its immutable content, the owner, and which clients have recently cached the object.

The service maintains a *client relationship graph* that models interest similarity between clients regarding content. Each node in this graph corresponds to a client and each edge between nodes indicates that the corresponding clients have similar interests. Based on findings in [4], we select the client that issues a request for an object with a probability *NAP* (Neighborhood Access Probability) from the object's owner or the owner's neighbors. Otherwise, we select a client uniformly at random from all clients.

When the service responds to a request from a client, it includes a caching directive. A directive denotes the set of clients at which the content should be cached. If empty, the receiving client does not need to cache or push the requested content. Otherwise, the client caches the content (if the client is in the list) and pushes the content to the other clients in the list, if any. Provided sufficient capacity, the receivers of pushed content cache it. A client that caches an object informs the service so the service can update the object's metadata.

3.2 Client-side Caches

Clients maintain a cache for their own use or to serve other clients. We suppose clients are non-malicious and follow the caching protocols described in this paper. Addressing threats related to data corruption can be achieved by caching data along with signed digests; dealing with other threats is the subject of future work.

The cache has a limited capacity. A client decides which objects to evict when its cache is full. Such eviction policies can be Least Recently Used (LRU), Least Frequently Used (LFU), or any other (e.g., ARC [8]).

When a client wants to access an object, it first checks its own cache. If absent from its cache, it sends a request to the service. The service responds with either the object's content or with a list of locations of other clients that have recently cached the object. In the second case, the client contacts these locations (one by one) to fetch the content. Clients receiving a "side-load" request check their caches and respond either with the object's content or with an error. If a client does not provide the object within a reasonable time, then the requesting client informs the service and tries another location, if available. If no client could be contacted, the object is requested directly from the service. Upon receiving the object, the client executes the caching directive.

4 Cache Placement Strategies

In this section, we present four cache placement algorithms. The first three are intended for baseline comparisons; the last one leverages social connections in order to improve individual hit ratios.

The objective of the *minimalistic* scheme is to minimize load on the server. To do so, it keeps at most one copy of an object in the collective client caches. The scheme works as follows: When a request for an object is issued, the service checks whether the object was recently cached by another client. If not, then the service provides the object itself and issues a directive to the client to cache it. Once the client receives the data, it caches the object using the LRU replacement policy. Otherwise, the service provides the requesting client with a singleton list containing the client that has recently cached the object and provides an empty directive (*i.e.*, the client should not cache it).

The *opportunistic* scheme is closest to the scheme CDNs usually implement, albeit a cooperative version of it. When the service receives a request for a particular key, it checks to see if any clients have recently cached the object. If so, it sends this list to the client. If not, the service responds with the object itself. In both cases the service sends a directive containing only the address of the requesting client. Thus, once the client receives the object, it caches the object and uses the LRU eviction policy, if necessary.

In the *popularity-based* algorithm, the service estimates the popularity of objects according to access rate

and fills client caches with the objects of highest popularity. Otherwise, the algorithm is much the same as the opportunistic one. The service adds the current global access rate (number of accesses per second) to the response to the client. If the client needs to evict an object from its cache, it uses the LFU policy by relying on the access rates provided by the service.

The *proactive* algorithm leverages the client relationship graph by proactively pushing content to the requesting client and its neighbors. The algorithm works as follows: when the service receives a request it responds with either the object or a list of locations, as previously. The directive, however, consists of the requesting client, the owner of the object, and neighbors of the owner, excluding the clients that appear off-line or have recently cached the object already.

5 Workload

In this section, we describe how we generate the synthetic workloads we use in our simulations. Our target is to approximate workloads typically found in online social networks and services where clients interact by sharing content with each other. In typical services such as Facebook, the corpus of objects continually grows. We have also seen that the popularity of objects is not fixed—objects tend to become less popular over time.

In our simulation, the corpus is a list of keys, ordered by popularity, initially consisting of a single key. The popularity distribution is a Zipfian distribution with skew parameter α. New objects are added to the corpus with a rate that is approximately 1/30th of the aggregate client request rate. We obtained this ratio from [3]. Each object is assigned a random owner from the set of clients. When a new object is added, we select an existing object from the corpus according to the Zipfian distribution and insert the new object before the selected one.

When a client makes a request, we also rely on the Zipfian distribution to select the requested key. We call this model the *shifting popularity model* (SP) since the popularity of keys in the corpus decreases as new keys are inserted into it. We have found that this model closely corresponds to measurements at Facebook [5].

Each time a key and an object are inserted into the corpus, the key and object sizes are chosen according to the models provided in [3] for Facebook's key-value store. The inter-arrival request rate also follows the model presented in [3] and varies between a few milliseconds to tens of milliseconds.

Another aspect of our workload is the client relationship graph that dictates the access patterns of clients. Since our workloads are parameterized by the number of clients, we employ a model for social networks that enables us to generate synthetic social graphs of various sizes. These graphs have similar properties such as node degree distribution and clustering coefficient (see [11] for more properties) to those found in real social networks. There are various models for social graphs and we chose the modified *nearest neighbor graph model* described in [11] that, according to the authors' analysis, better approximates real social networks w.r.t. the previous properties. To derive the parameters of the previous model, we generated graphs of sizes equal to those of real world graphs found in [1] for various value assignments of the parameters. We then compared the generated graphs with the respective real world graphs w.r.t. their node degree distribution and clustering coefficient.

For each synthetic workload, we generated a client relationship graph G and fixed NAP (Section 3.1) to 0.8. This value comes from [4] and is consistent with client behavior observed in social networks such as Orkut, LinkedIn and others based on click-stream data. The graph G, along with the skew parameter α are given as input to the request generation procedure described in Algorithm 1.

Algorithm 1 Request generation algorithm

```
1: procedure GENERATE REQUEST(α, G)
2:     select a key k according to the SP model with parameter α.
3:     if p ≤ NAP for p ∈ [0,1] chosen uniformly at random then
4:         select the client v issuing the request uniformly at random
5:          from the object's owner and its neighbors.
6:     else
7:         select client v uniformly at random from all clients.
8:     end if
9:     return (v,k)
10: end procedure
```

6 Experiments

In this section, we compare the cache placement strategies described in Section 4. We assess key cache performance metrics through simulation on synthetically generated workloads as described in Section 5.

6.1 Setup

Three parameters describe the configuration of our experiments: *the number of clients* that issue requests to the service, *the cache capacity per client* measured in bytes, and *the skew parameter of the Zipf distribution* that determines the popularity distribution of objects. In each run

of the simulation, clients have the same capacity.

We run our simulations for varying configurations. In all configurations, the cache capacity per client is 100KB and the skew parameter is 1.1. We tried higher cache capacities and skews, but the trends are the same for each. In order to avoid the bias resulting from a cold cache and the initially small corpus, we begin each run with a 3 minute warm up phase. We then collect measurements for another 3 minutes. For each client c, we monitor the following:

- h_c, the number of requests it serves from its own cache (local hits),
- s_c, the number of requests which are served by another client (side-loads),
- m_c, the number of requests which are served by the service (client cache misses),
- b_c, the average bandwidth (over the length of the experiment) spent serving and pushing content to other clients.

Three metrics are presented in the evaluation:

- The average local cache hit ratio: $\sum_c \frac{h_c}{h_c+s_c+m_c}/n$
- The collective (global) hit ratio of the client caches: $\frac{\sum_c h_c+s_c}{\sum_c h_c+s_c+m_c}$
- The average bandwidth spent per client for serving and pushing content: $\sum_c b_c/n$

where n is the number of clients. In our measurements, the opportunistic and popularity-based algorithms perform similarly, and the graphs often overlap. For clarity we have left out the error bars—they become negligible when the number of clients is large.

6.2 Local hit ratio

In Figure 1a we show how the average local hit ratio varies with an increasing number of clients. We see that as the number of clients increases, the local hit ratio increases as well for all but the minimalistic algorithm. This is a consequence of the workload model that we use: popular objects are likely to be requested by the same neighborhood of clients, and as the number of clients increases the neighborhoods get larger, which drives up the average local hit ratio. This is not so for the minimalistic algorithm as it maintains only a single copy of each object in the collective cache. The proactive algorithm benefits most from an increased number of clients since it proactively creates copies of the accessed objects on clients that are more likely to access them. Results not shown here indicate that as the capacity per client increases, the benefits are even larger. The proactive algorithm remains the best one.

6.3 Global hit ratio

The global hit ratio is the hit ratio of the collective cache, and ultimately determines the load on the service. Figure 1b shows the global hit ratio as the number of clients increases. More clients means a larger collective cache and thus a higher global hit ratio. The hit ratio does not hit 100% because there is a steady stream of new objects that are being introduced.

The minimalistic algorithm exploits the collective capacity of the caches best. The other algorithms create multiple copies for popular objects and thus evict more objects than the minimalistic algorithm. (Although not considered in these experiments, churn would negatively affect the minimalistic algorithm more than the others as only one copy of an object is maintained.)

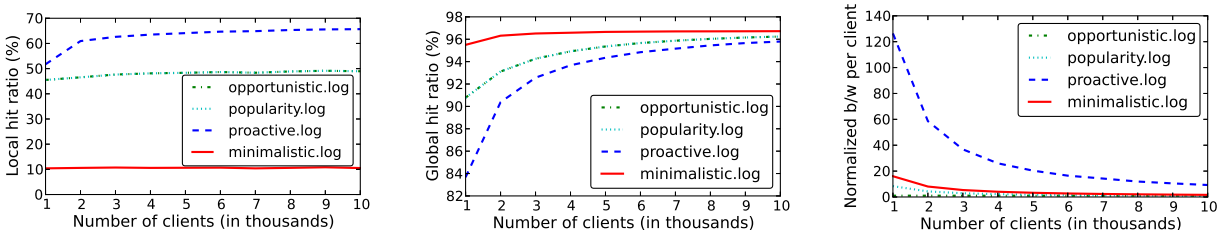
The opportunistic and popularity-based algorithms do worse but appear to catch up with an increasing number of clients. As the number of clients grows, so does the number of neighborhoods and the neighborhoods' average size. Due to our workload model, it becomes increasingly likely that an object is cached in the neighborhood of its owner. As a result, opportunistic and popularity-based algorithms catch up with the minimalistic one.

The proactive algorithm performs worst with respect to global hit ratio: it suffers from a higher eviction rate caused by pushing. However, it also benefits from more and larger neighborhoods as the number of clients grows.

6.4 Client bandwidth

In Figure 1c, we see the average outgoing bandwidth per client as a function of the number of clients normalized for the opportunistic algorithm. The opportunistic algorithm uses the least outgoing bandwidth. In the minimalistic algorithm, the clients do much side-loading because of two reasons. First, the number of cached objects is higher than for the other algorithms, resulting in a larger opportunity for side-loading. Second, there is at most one cached copy of each object, and thus most requests require a side-load.

At the other extreme is the proactive algorithm, which pushes many objects proactively, something the other algorithms do not do at all. From our experiments, we conclude that our current implementation of proactive pushing is more expensive than opportunistic side-loading. However, because the local hit ratio improves with the number of clients and the eviction rate decreases due to the higher global hit ratio, the overhead of pushing drops.



(a) Local hit ratio (b) Global hit ratio (c) Local outgoing bandwidth

Figure 1: Local hit ratio per client (1a), global hit ratio of the collective cache (1b) and normalized bandwidth per client (1c) for skew parameter 1.1 and a capacity of 100KB per client for a varying number of clients.

While expensive compared to the opportunistic algorithm, in absolute terms the overhead is reasonable. For 10,000 clients the average bandwidth for the proactive scheme is about 9.3 KB/s. The overhead for the opportunistic and popularity-based schemes is $\sim$ 800 B/s, while $\sim$ 1.6 KB/s for the minimalistic scheme.

7 Conclusions

In this work, we investigate the effects of cache placement on the performance and cost of a collective cache built on the clients of an OSN. We propose a caching scheme that employs relationship information between clients. Simulations of our scheme using synthetically generated workloads suggest that client relationship information improves the client's perceived hit ratio at a reasonable overhead.

The proactive scheme that we present in this paper presents promising results. There are several open questions that remain to be answered however. We need to assess the efficiency of this approach in the presence of client churn, its impact on latency, and the additional computational and memory costs required to maintain the caching metadata. In particular, we are interested in knowing whether social proximity is detrimental to latency when compared to schemes that rely on geographic locality. We also wish to determine whether we can design a hybrid strategy that combines the best features of the proactive and popularity schemes. In particular, is it possible to reduce the communication overhead of the proactive scheme without negatively impacting the local hit ratio?

References

[1] `http://snap.stanford.edu/data/`. [Online; accessed 2012].

[2] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, London, England, UK, 2012. ACM.

[4] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet Measurement Conference*, IMC '09, pages 49–62, Chicago, Illinois, USA, 2009. ACM.

[5] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. Li. An analysis of Facebook photo caching. In *Proceedings of SOSP 2013*, 2013.

[6] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 213–222, Monterey, California, 2002. ACM.

[7] M. Marcon, B. Viswanath, M. Cha, and K. P. Gummadi. Sharing social content from home: a measurement-driven feasibility study. In *Proceedings of the 21st international workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '11, pages 45–50, Vancouver, British Columbia, Canada, 2011. ACM.

[8] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[9] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1587–1596 vol.3, 2001.

[10] L. Ramaswamy, L. Liu, and A. Iyengar. Cache clouds: Cooperative caching of dynamic documents in edge networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS 2005*, pages 229–238, 2005.

[11] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y.

Zhao. Measurement-calibrated graph models for social network experiments. In *Proceedings of the 19th international conference on the World wide web*, WWW '10, pages 861–870, Raleigh, North Carolina, USA, 2010. ACM.

[12] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 203–213, London, UK, 2002. Springer-Verlag.

[13] D. A. Tran, K. Nguyen, and C. Pham. S-clone: Socially-aware data replication for social networks. *Computer Networks*, 56(7):2001 – 2013, 2012.

[14] J. Wang. A survey of web caching schemes for the Internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.

[15] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 16–31, Charleston, South Carolina, USA, 1999. ACM.

[16] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram. Maygh: building a CDN from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 281–294, Prague, Czech Republic, 2013. ACM.