Proactive cache placement on cooperative client caches for online social networks

Stavros Nikolaou, Robbert Van Renesse, and Nicolas Schiper, *Cornell University*

Abstract—This paper investigates cache placement on a cooperative cache built from individual client caches in an online social network or web service. We use a service that maintains a mapping between content and the clients that cache it, and propose cache placement schemes that leverage relationships between clients (for example, social links) and workload statistics, proactively placing content on clients that are likely to access it. We evaluate efficacy through simulation, comparing our schemes against commonly used cache placement algorithms as well as optimal placement. We synthesize a workload to match characteristics of online social networks. Simulation results of our proposed caching schemes impose moderate network overhead and show considerable improvement to the client's cache hit ratio, even under churn.

Index Terms—Cooperative caching, cache placement, social networks

1 Introduction

The increasing popularity of today's content-sharing web services such as Online Social Networks (OSNs), photo-sharing websites, and video-on-demand systems is leading to vast amounts of generated content. Services need to either rely on Content Delivery Networks (CDNs) and cloud storage to meet the increasing demand, or build, deploy, and manage their own content delivery systems. Both approaches are expensive and thus are often only accessible to large companies.

An alternative is to deploy cooperative caching [1], [2], where clients of the service act as caches and serve content to each other in a peer-to-peer (P2P) fashion. Cooperative caching has multiple benefits: First, it offloads the service, thus mitigating the cost of content delivery. Secondly, it helps with flash crowds [3], since a flash crowd will provide the service with a proportionally large cache capacity to deal with the spike in demand. Thirdly, cooperative caching schemes can substantially reduce access latency since content resides close to the clients themselves. Finally, it improves network resource utilization by distributing bandwidth usage to many client-to-client connections, offloading the service's backbone network.

Cooperative caching has been around for some time, but it has recently gained renewed interest for various reasons. First, technology has become available that enables easy deployment of such schemes. Maygh [2] is an example system that leverages non-intrusive technologies such as Flash and WebRTC for browser-to-browser communication to show the feasibility and benefits of cooperative caching for today's services. Secondly, there is increasing availability of relationship information between clients such as social links, users subscriptions to other users' updates, etc. Such information can provide the service with better insights into access patterns, which can be leveraged for efficient cache placement strategies.

This work focuses on how to better organize the client caches in a cooperative cache setting and studies the cache placement problem for a cooperative cache built on the clients of an OSN or similar content-sharing services. Compared to our previous work [4], we have developed improved strategies, address the effect of client churn, use more realistic workloads to evaluate our strategies, and consider privacy issues that arise in cooperative caching.

We consider a system model similar to that of Maygh. Each client maintains a cache of limited capacity for caching content and can communicate with other clients. The service has a growing corpus of objects that can be requested by clients and maintains an approximate mapping between objects and the clients that cache those objects.

Because strategically placing cached content can significantly affect performance and overhead in such a cooperative environment [5], we investigate how cache placement affects hit ratio and load perceived by the service and the clients. We make the following contributions:

- We propose two novel cache placement strategies that take advantage of known relationships between clients (for example, social links) and the workload on the service. Both schemes proactively create cached copies of content on clients that are more likely to access it based on those relationships. One of these strategies allows tuning proactivity, thus controlling the approach's tradeoff between benefits and costs;
- We evaluate our approaches and compare them with three common placement strategies. We also compare them with optimal placement schemes that assume full knowledge of the workload. Our evaluation uses simulation on synthetically generated graphs and workloads that match certain characteristics of OSNs [6], [7];
- We evaluate how our approaches work under churn.

Our findings suggest that we can substantially improve clients' hit ratios over common strategies at modest overheads. We show that, under certain assumptions, we can get a near-optimal client hit ratio.

2 System Model

In this section, we describe the model we assume for our cooperative cache. The model consists of two main components: a service and the set of its clients. We discuss these components and their interactions, and we consider security and privacy implications of cooperative caching.

2.1 The Service and its Clients

The main role of the service is twofold: First, in response to a client request, it either serves the contents of an object, or the location where the object is cached. Secondly, it determines which clients should cache the objects. We assume that objects are immutable; mutable objects are modelled as a sequence of versions, each being an immutable object. Each object has an owner—one of the clients of the service. We identify objects via *keys* that the service maintains in a key-value store along with the corresponding immutable content, the owner, and the set of clients that have recently cached the objects. Recency is here considered with respect to some time window determined by the service during which a client has been directed to cache an object. These directives are described in greater detail later in this section. The service tracks which clients are online by monitoring their most recent activity with respect to this time window.

Since we are focusing on OSNs, we assume that the service maintains a *client relationship graph* which encodes client subscriptions to the objects of a particular owner. In this graph, nodes represent clients of the service and edges represent client subscriptions. Such a graph can effectively model an OSN where client subscriptions represent mutual interest in the objects published by the corresponding clients. We assume the service either has access to such a graph or can construct it by enabling its clients to subscribe to each other's objects. The exact details of how the service obtains the client relationship graph are outside the scope of this work. Our workload model is based on [8]: objects are accessed with a heavy-tailed distribution. The client accessing such an object is selected as follows: with probability *NAP* (*Neighborhood Access Probability*) the client comes from the *neighborhood* of the owner of the object, that is, from either the owner itself or one of its neighbors in the client relationship graph. Otherwise (with probability $1-NAP$), the client is selected uniformly at random from all clients excluding the ones in the neighbourhood of the owner.

The service interacts with the clients using what we call *cache directives*. A cache directive is a message that the service sends to a client either as a *reply* to a client's request or as an *unsolicited cache directive* and has the following format: (ID, C, D, L). ID is the identifier of an object (this can be a collision-resistant hash of its content), C is a boolean used to indicate whether the receiving client should cache the object or not, D is the content of the object (can be $\perp$ to indicate that the content is not included in the message), and L is a set of cache locations, that is, a set of clients.

If a cache directive is a reply to a client's request then ID holds the identifier of the object requested. If $D \neq \perp$ then the service is providing the object's content to the client and $L = \emptyset$. If $C = \texttt{true}$ then the service directs the client to store (ID, D) in the the client's cache. Conversely, if $C = \texttt{false}$, the client should not cache the object. If $D = \perp$ then the content is not provided in the reply and the receiving client needs to fetch it from one of the locations in L, and thus $L \neq \emptyset$. To do so, the client will issue a *side-load* request to locations in L. We describe side-load requests in Section 2.2.

If a cache directive is an unsolicited one there are two cases: First, the directive is of the form, $(ID, \texttt{false}, \perp, \emptyset)$, in which case the receiving client must evict object ID from its cache if it is there. Second, the directive is of the form $(ID, \texttt{true}, \perp, L)$, where $L \neq \emptyset$ and the receiving client must fetch object ID from one of the locations in L and cache it even though that client did not explicitly requested it. When the service issues an unsolicited cache directive of the second form, we say that the service *pushes* object ID to the receiving client.

2.2 Client-side Caches

Each client is equipped with a cache of limited capacity that it can use for serving client requests. A client may evict an object from the cache either when it receives an eviction directive or when it has to cache an object and its cache is full. In the latter case, the client decides which objects to evict using some eviction policy such as Least Recently Used (LRU), Least Frequently Used (LFU), or Adaptive Replacement Cache (ARC) [9].

A client can interact with the service and other clients using three types of messages: `request`(ID), `side-load`(ID) and `cache-update`(ID, U). When a client requires access to an object, ID, that does not reside in its cache, the client sends a `request`(ID) message to the service. The service responds with a reply cache directive like those described in Section 2.1. If the response has a non-empty location list, the requesting client sends `side-load`(ID) messages to each of the clients in the location list, starting with the first one. A client receiving a side-load request for some object, ID, checks its cache and if ID is there, responds with the content of the object; otherwise it responds with an error. If the requesting client receives an error or times out while waiting for a response from a location, it sends a `cache-update`(ID, U) message to the service. The U component of the previous message is a list containing the clients that did not respond in time with the content of the object requested. If the service receives such a cache-update message, it updates its meta-data regarding the clients caching object ID. If none of the clients on the location list provided by the service is able to serve the object, the client obtains the object from the service directly by sending another `request`(ID) message. Once the requesting client obtains the content of object ID, it executes the cache directive provided by the service according to C's value.

2.3 Integrity and Privacy

Cooperative caching has significant security concerns. As clients are serving objects, malicious clients can modify them. This, however, is easy to prevent with digital signatures [10].

A more difficult problem concerns the privacy of clients. If the server directs client c to obtain an object from client

c', then c and c' learn something about one another, and in particular may learn something about one another's interests. In a previous incarnation of our system [4], it was indeed trivial to learn this. We have made this significantly harder in our new approaches.

The main idea is the following: when a client c' receives a request for an object from another client c, c' cannot know if c requested the object from the service or whether the service sent c an unsolicited directive to cache the object. Vice versa, c cannot know if c' directly requested the object in its cache either. This gives both c and c' a level of *plausible deniability*. However, c' still learns that c is in the neighborhood of a client that requested the object. In order to increase the amount of plausible deniability, the service can sometimes select random clients to cache objects, similar to what was suggested for Maygh [2].

3 Cache Placement Strategies

In this section, we present various cache placement algorithms. The first three are intended for baseline comparisons. Next we present the proactive algorithms we have designed, leveraging social connections in order to improve individual client hit ratios. Then we present optimal versions of each algorithm, assuming knowledge of the future.

3.1 Baseline Approaches

Opportunistic Approach

The *opportunistic* scheme is a commonly used approach for CDNs and web caches. Maygh, BitTorrent, and Gnutella all use a variant of this approach. Upon receipt of a request, the service checks to see if there are clients caching the object. If so, the service responds with the locations of those clients. If not, the service provides the object itself. In either case, the service directs the client to cache the object ($C = \texttt{true}$), using LRU eviction if its cache is full.

Minimalistic Approach

The objective of the *minimalistic approach* is to minimize load on the server. To do so, it tries to keep at most one copy of an object in the collective cache of the clients. The scheme works as follows: When the service receives a request for an object, the service checks whether the object was recently cached by another client. If so, the service provides the requesting client with a singleton set L containing the client that has recently cached the object and sets directive C to `false` (*that is*, it requests the client not to cache the object). Otherwise, the service sends a directive to the client containing the content and sets C to `true`. In that case the client caches the object. If its cache is full, then LRU replacement applies.

Minimalistic*

The minimalistic approach has two main problems. First, a client that has a copy of a highly popular object becomes a hotspot since it must serve all requests for the object. Second, in case of churn many client-to-client loads may fail. Minimalistic* is a variant that only tries to minimize the number of copies held for unpopular objects.

When the service receives a request for an object ID from a client c, it first checks to see if there is a client c' that has recently loaded the object. If not, the service returns the object to c with the directive to cache it. If there is a client c', the service sends to c a directive $(ID, \texttt{true}, \perp, \{c'\})$, directing c to load the object from c' and add the object to its cache. Upon receipt, c informs the service. If the popularity of ID is below a configured threshold, then the service directs c' to evict the object. The service approximates the popularity of an object by the fraction of requests issued for that object in a configured window of time.

For small thresholds, the algorithm approximates the opportunistic approach, because it is more likely that a copy of the requested object will be created on each request (for threshold value 0 they become identical). For large thresholds, the algorithm approximates the minimalistic algorithm.

3.2 Proactive Approaches

Basic Proactive Approach

The basic *proactive* algorithm leverages the client relationship graph by proactively pushing content to the neighborhood of the content owner. We here define the *neighborhood* of the content owner, or simply client, as the set containing the client itself as well as its neighbors in the client relationship graph.

When the service receives a request, it responds with either the object or a list of locations, L, as in the opportunistic scheme. The response is always a caching directive, that is, $C = \texttt{true}$. In addition, the service selects the $\lceil r \times |N_{owner}| \rceil$ most recently active clients from the owner's neighborhood, where $r \in (0, 1]$, the *replication factor*, is a parameter of the algorithm and N_{owner} is the set of neighbors of *owner*. The service issues a directive to each of them to side-load and cache the object, excluding any clients that appear off-line or have recently cached the object already. Clients that do not have enough space left in their cache use LRU eviction.

Common Neighbors Proactive

We also propose a variant of the previous approach that we call the *common neighbors proactive* scheme. With this variant, the subset of the owner's neighborhood to which the object is pushed is the intersection of the owner's neighborhood and the neighborhood of the client requesting the object.

3.3 Space and Time Complexity

The space complexity of the previous approaches is the state required by the service to keep track of online clients and their cached objects. All approaches require this meta-data and thus the space complexity for all approaches is $O(n \cdot m)$ where n is the number of clients of the system and m the total number of objects. Given an object identifier, the service needs to provide the set of clients that cache it. It must, therefore, keep a mapping from objects to caching clients. We assume that $m \geq n$, which is reasonable for social networks since in most social networks users typically need an account to access data on the service and thus at least a profile object must exist per user. Notice that due to the previous assumption, the additional client relationship graph required by the proactive approaches—which would typically require $O(n^2)$ space—is included in the previous bound. The size of the previous state increases substantially as the number of clients and objects increases, and thus additional measures need to be taken for keeping the state size manageable in popular social networks. Such measures may include garbage collection of old and/or unpopular objects and sharding of the service state across multiple machines, for example, by employing consistent hashing, geo-spatial information etc.

We consider time complexity with respect to the number of messages that need to be sent until the client receives the object it requested. We briefly describe the worst case analysis of a client's request for each strategy. Each approach requires sending a message to the service to get a location of the cache, if it exists, or the object content otherwise. If a location is known, another message needs to be sent to that cache. From this point forward each approach may differ on the number of messages sent. In the minimalistic approach, a request will always be served after at most three messages. The first two messages are as above. The last one is sent to the service for retrieving the object when the provided location is no longer available, either due to time-out, failure, or due to the object no longer being cached at the location. The remaining approaches are similar with respect to the number of messages. This number is bound by the number of locations provided by the service, that is, $O(n)$. Notice that, by our model description, each time a client detects a cache location unable to provide the object, it has to inform the service via another message. This communication, however, is not crucial for obtaining the object and thus can be performed in the background. The previous bound can be very impractical and thus, in a real deployment, we could employ some optimizations for reducing the number of messages sent. Such optimizations might include trimming the service response to a fixed number of locations and contacting multiple locations in parallel, at an additional bandwidth cost.

Table 1 summarizes the previous discussion.

Algorithms	**Space**	**#Messages**
Minimalistic	O(n*m)	3
All others	O(n*m)	O(n)

TABLE 1: Space and message complexity of the cache placement strategies. Here n is the number of clients and m the number of objects.

The previous discussion suggests that minimalistic is the best algorithm since it performs better with respect to message complexity and equally well with respect to the memory needed. In practical systems however, client specific metrics like latency perceived and bandwidth spent to support the collaborative cache that offloads the service are more important. In addition, the cache described in Section 2 needs to gracefully handle the churn that is inherent in the peer-to-peer nature of the system. As we later see in the evaluation of the approaches (Section 5), the minimalistic strategies do not handle churn well. Finally, we note that all approaches scale similarly, though the proactive strategies do require additional effort for managing large scale client interaction graphs.

3.4 Optimal Variants

For comparison purposes, we have also implemented optimal variants of all previous algorithms. These variants assume knowledge of the future, which is provided in the form of an oracle that knows the trace of requests a priori.

For the minimalistic and opportunistic approaches, the oracle is used during the eviction process when the cache of the clients is full. In other words, clients implement Belady's optimal algorithm for cache replacement [11].

For the proactive approaches, the oracle is used for optimal eviction as well as for optimal pushing. The optimal proactive schemes push copies only to those clients that will be requesting the object in the future.

There are two flavors of the optimal proactive algorithm. The first one, which we call *globally optimal proactive*, has the client push the object to all clients that will request it in the future (and potentially outside of the client's neighborhood), excluding those that already cache it. The second flavor is called *neighborhood only optimal proactive* and works as the previous one with the restriction that copies are pushed only to clients that will request the object and are in the neighborhood of the owner.

4 Workload

In this section, we describe how we generate the workload used in our evaluation (Section 5). Our main focus is to capture workload characteristics of realistic OSNs where clients may share and subscribe to each other's content.

One of the workload aspects we are interested in is the corpus of objects requested. We observe that the corpus of objects in OSNs continuously grows as time passes (for example, due to photos and posts uploaded by users). As a consequence, the relative popularity of objects decreases over time: old content becomes stale and unpopular, whereas new content gets more attention.

To capture these observations, we assume a corpus represented as a list of keys ordered by the objects' popularities. We insert new objects in this corpus at a rate of approximately 1/30th of the aggregate request rate. We consider key insertions to play the role of writes and all clients' requests to be the reads to the service. This read-heavy workload is commonly observed in OSNs and approximates Facebook's key-value store [6] with respect to the read/write ratio.

When a new object is added to the corpus, its popularity (that is, its rank in the corpus list) is picked from a Zipfian distribution with skew parameter α. Once the object's rank is decided, its respective key is inserted in the corpus, increasing all equal or higher ranks by one. For each new object, an owner is selected uniformly at random from the set of clients. The key and content sizes of the newly inserted object are selected from the distributions provided by [6], reflecting Facebook's key-value store and matching closely the object size distribution at Facebook [12]. Each time a new request is issued, we select the object according to a Zipfian distribution with skew parameter α.

According to the model above, the popularity of objects decreases as new objects are added in the corpus. We name this model the *shifting popularity model* (SP). This model was found to closely correspond to measurements performed on Facebook's photo corpus [12]. The initial size of the corpus determines the speed at which the popularity of objects change. We revisit this issue in Section 5.

The time between client requests follows a Generalized Pareto distribution [6]. Request inter-arrival time is concentrated around $\sim 500 \mu s$.

The last component of our workload generator is concerned with the client relationship graph that is used for selecting the client that issues a request for a given object. The specifics of the selection have been described in Section 2.1. For this work, we assume that the client relationship graph is static. This assumption simplifies cache location selection and generation of cache directives.

We leverage a graph model to synthetically generate graphs of any given size that share common graph characteristics such as node degree distribution, clustering coefficient, and others, with real social networks. There is a large number of graph models for social networks [13], [14], [15]. In this paper, we chose the modified *nearest neighbor graph model* [7], $G(n,u,k)$, because this model can generate graphs that approximate real social graphs best. The generation proceeds in steps. The model takes three parameters: 1) the total number of nodes, n, 2) a probability, u, that determines at each step if a new node is added or if a pair of 2-hop neighbors are connected, and 3) the number of node pairs, k, that are connected on a node addition. We derived the parameter values by fitting the nearest neighbor model to a set of real world graphs found in [16]. Our objective function for the fitting process takes into account the similarity of the generated graphs to the real ones with respect to node degree distribution and clustering coefficient.

Our request generation procedure takes a client relationship graph G, the NAP probability, and the skew parameter α as input and is described in Algorithm 1.

Algorithm 1 Request generation algorithm

```
 1: procedure GENERATE REQUEST($G$, $\alpha$, $NAP$)
 2:     select a key $k$ according to the SP model with parameter $\alpha$.
 3:     if $p \leq NAP$ for $p \in [0,1]$ 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:             excluding the owner and its neighbors.
 9:     end if
10:     return $(v,k)$
11: end procedure
```

5 Evaluation

In this section, we compare the cache placement strategies described in Section 3. We assess key cache performance metrics through simulation on synthetically generated workloads as described in Section 4. There is a trade-off between the fidelity of a simulation and the scale at which we can run such a simulation. In order to run the simulations in a reasonable amount of time, the number of clients and the number of objects in the corpus in our experiments are significantly smaller than in a popular social networking site. A large service will still require sharding and multi-level caching such as used at Facebook [12], but this section will demonstrate that medium-scale services can significantly benefit from proactive cooperative caching. Existing scaling techniques like those presented in [2] may be used for porting the ideas illustrated here to systems of larger scale.

5.1 Setup

In our experiments, we vary the number of clients n from 1 to 10,000 and keep the cache capacity per client, c, constant at 5MB. Many browsers, particularly on mobile devices, impose a 5MB limit on their HTML5 cache sizes [17]. We did experiment with larger cache sizes and found similar trends as reported below. The skew parameter of the Zipf distribution, α, is kept constant at 1.1. We obtained this parameter value from [18], where a similar decreasing popularity mechanism is described and found to approximate a popular social network. Experimentation with larger skews also result in similar trends.

For a range of n, we generate a client relationship graph G according to the nearest neighbor model (Section 4) with parameters $u = 0.96$ and $k = 1$. Then for each graph G, we generate a workload of 20 million requests using Algorithm 1 with *NAP* fixed to 0.8. The *NAP* value comes from [8] and is consistent with click stream behavior observed in social networks such as Orkut, LinkedIn, and others. (In Section 5.6 we investigate the effect of using a small *NAP*.) The size of the trace corresponds to 150 minutes of execution using the request rate discussed in Section 4.

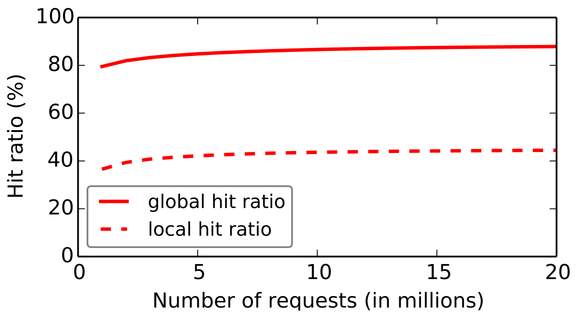
Even though our workload implies the same number of requests for different numbers of clients, the results do not change. This is because after the warm up phase, and given the large initial size of the corpus, we reach a steady state behavior with respect to popularity of objects requested by each client. Thus, additional requests issued for smaller numbers of clients have negligible impact on the metrics we are interested in.

For each client c, we monitor the following:

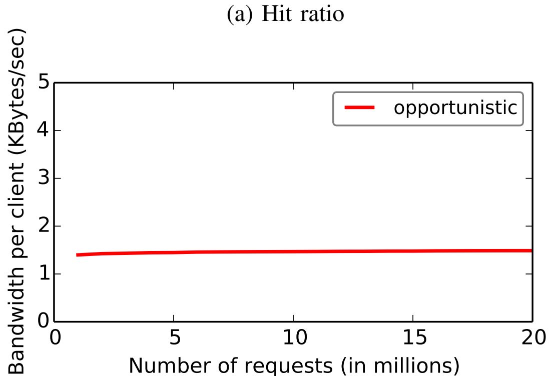
- h_c, the number of requests it serves from its own cache (local hits),
- s_c, the number of requests that are served by another client (side-loads),
- m_c, the number of requests that are served by the service (client cache misses),
- b_c, the average bandwidth (over the length of the experiment) used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

The metrics examined in this evaluation are the following:

- 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: $(\sum_c h_c + s_c)/(\sum_c h_c + s_c + m_c)$
- The average bandwidth spent per client on serving content: $\sum_c b_c/n$



(a) Hit ratio



(b) Local outgoing bandwidth

Fig. 1: Average local hit ratio per client and global hit ratio of the collective cache (a), and average bandwidth per client (b) for the opportunistic approach with 10,000 clients as time passes.

A local cache hit means that the client can serve an object out of its own cache. A global cache hit means that the server is invoked but the object is served from the cache of a client. A miss in both means that the server has to serve the object's content.

An important consideration in our experiments is bias caused by the initial conditions of the simulation, in particular the initial size of the corpus. The fraction of objects that can fit in the collective cache is inversely proportional to the size of the corpus. In addition, the selection probabilities of each rank in the Zipf distribution change substantially as the corpus size increases for the first few million objects. To reduce the effects of an initially small corpus, we initialize the corpus with 10 million objects.

In Figure 1 we observe how the previously described metrics evolve over time for the opportunistic approach with a configuration of $10,000$ clients. The corpus is initialized as previously mentioned and the client caches start empty. The x-axis shows the number of requests issued to the service from the beginning of the experiment. A simulated second contains $\sim$2,000 requests. As the figures show, each metric stabilizes after approximately 10 million requests. This stabilization behavior is similar for all cache placement algorithms discussed in the paper.

In the following simulations, we begin each run with a 10 million request warm up phase ($\sim$75 simulated minutes), and then collect measurements for another 10 million requests.

We now summarize the various parameters that affect our simulations. We categorize them in three groups: First, there are those related to the cache placement algorithms such as the minimalistic*'s popularity threshold and the replication factor of the basic proactive approach. Both determine how many copies of an object should be cached in the collective cache, but they target different sets of client caches.

Second, there are the parameters associated with the *workload generation*. These parameters are: i) The number of clients, or size of the client relationship graph, which ultimately determines the size of the system and collective cache. ii) The neighborhood access probability *NAP*, set to 0.8, which determines the user requesting a particular key. iii) The percentages of read and write requests in the workload set to 95% read and 5% write requests for all experiments. iv) The skew of Zipfian distribution that selects the next key to be requested, fixed at 1.1. v) The key and value sizes (in bytes) for each object that are determined by the Generalized Extreme Value and Generalized Pareto distributions respectively. The former has parameters $\mu = 30.7984$, $\sigma = 8.20449$, and $k = 0.078688$ and the latter has $\theta = 0$, $\sigma = 214.476$, and $k = 0.348238$. Both distributions are taken from [6]. vi) The nearest neighbor model's parameters, that is, the probability $u = 0.96$, and the number of pairs $k = 1$, used for generating client relationship graphs that mimic real social networks.

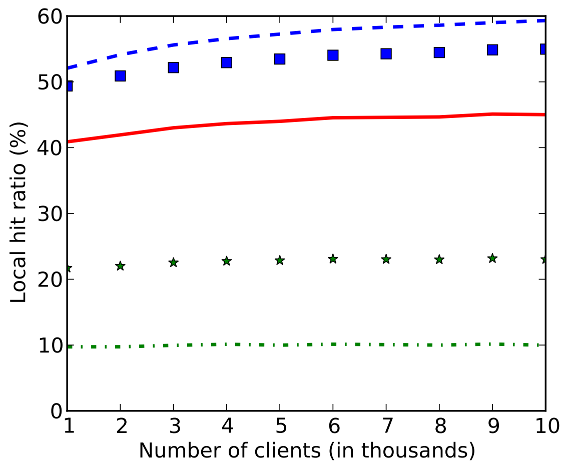
Finally, there are the parameters that are specific to the simulations we ran. These concern the cache size of each client, which is set to 5MB for all clients, and the session window. The latter is expressed in number of requests received by the service before a client who has not issued any requests is considered as offline. The session is used to tune the churn in the experiments presented later in the section.

5.2 Base case comparison

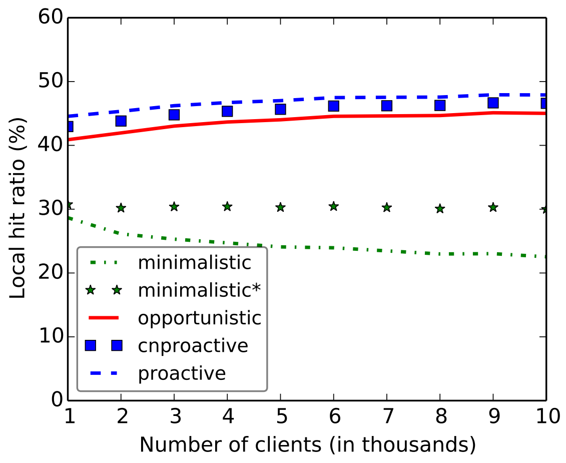
In this section, we compare the strategies presented in Section 3. In the simulations, clients of the service remain online throughout the experiment once they have opened a session (that is, no churn). Clients start issuing requests at different times however. Nevertheless, if there is a client caching the requested object, then a side-load will be successful.

Some of the approaches described in Section 3 are parameterized. Minimalistic* uses a threshold parameter that determines whether it creates a new copy or simply moves the requested object from the caching client to the requesting one. For all configurations discussed in our evaluation, we set the threshold to 0.1.

The proactive approach is parameterized with a replication factor that determines the fraction of the neighborhood of the owner of the requested object to which the object will be pushed. In the following experiments, all runs of the proactive algorithm have a replication factor of 1, that is, the requested object is pushed to all neighbors excluding clients that already cache the object or are off-line. See Section 5.4 for more information on how the replication factor affects the proactive approach performance and cost. Our proposed proactive variant "common neighbors proactive" is denoted as "cnproactive" on all figures that follow.

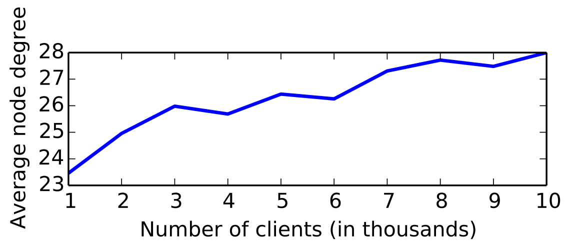


(a) Local hit ratio

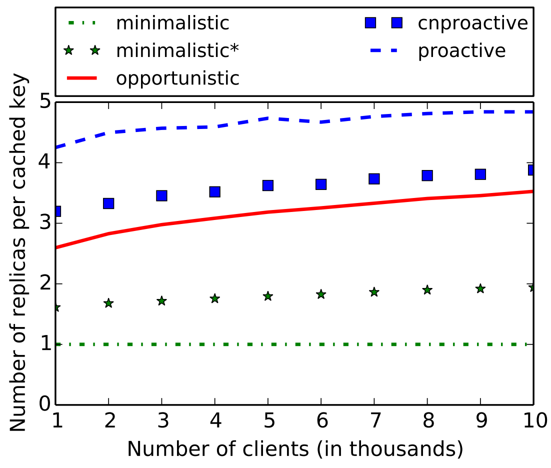


(b) Local hit ratio with churn

Fig. 2: Average local hit ratio per client (a) and local hit ratio running under a churn rate of 0.1 (b) for a varying number of clients.



(a) Average node degree of synthetically generated graphs.



(b) Average number of replicas per cached key.

Fig. 3: Average node degree (a) and average number of replicas per key (b) for a varying number of clients.

5.2.1 Local hit ratio

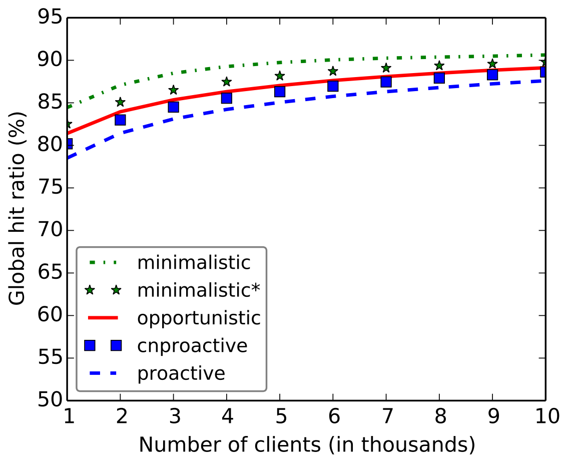
In Figure 2a, we show how the average local hit ratio varies with an increasing number of clients. We see that as the number of clients increases, local hit ratio increases slightly for all but the minimalistic algorithm. This is a consequence of the workload model that we use and of the way that the client relationship graph grows as the number of clients increases. Each object is likely to be requested by the neighborhood of its owner because the NAP probability is 0.8 in our experiments. As the number of clients increases, the neighborhood size grows as depicted in Figure 3a. In Figure 3b, we observe that an increase in the number of clients results in a higher number of object replicas per key.

Since the popularity of each object in each configuration remains the same, but the number of users caching it increases, there is a greater number of users that are likely to find it in their cache, in turn resulting in an increased local hit ratio. Note that this slight increase is not observed for the minimalistic approach because it only creates a single cached copy per object regardless of its popularity or of the number of clients requesting the object.

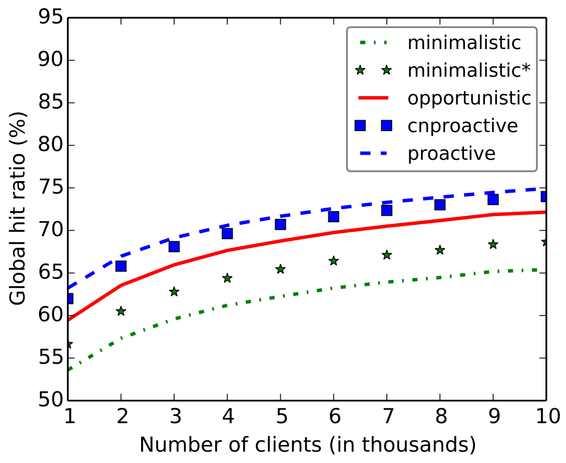
The proactive algorithm performs best because it creates copies of the accessed object on clients that are more likely to access them. By targeting the right clients, the proactive algorithm increases substantially the likelihood that a future request for an object will be found in the requesting client's cache. With respect to the local hit ratio, the remaining approaches are ordered according to the number of replicas they create per key (Figure 3b). The more replicas for an object, the more clients caching it and thus the higher the local hit ratio. The proactive approaches perform considerably better, with more than 10% improvement over the reactive techniques (the opportunistic, minimalistic, and minimalistic* algorithms).

5.2.2 Global hit ratio

The global hit ratio is the hit ratio of the collective cache and heavily influences load on the service. Figure 4a shows 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 reach 100% because there is a steady stream of new objects that are being introduced. Although the proactive and common neighbors proactive approaches



(a) Global hit ratio



(b) Global hit ratio with churn

Fig. 4: Global hit ratio (a) and global hit ratio running under a churn rate of 0.1 (b) as the number of clients increases.

perform worse than the other techniques, their respective hit ratio is only ∼4% lower than the minimalistic algorithm.

The minimalistic algorithm exploits the collective capacity of the caches best because it creates no duplicates. As we will see in Section 5.5, the performance of the minimalistic approach with respect to this metric is very sensitive to churn as only one copy of an object is maintained. The other algorithms create multiple copies for popular objects and thus have less cache capacity for storing additional objects. This is shown clearly in Figure 5, which depicts the ratio of objects cached at the end of the simulation over the total number of objects cached during simulation. The larger the number of replicas that an algorithm creates, the fewer objects that can be stored in the collective cache.

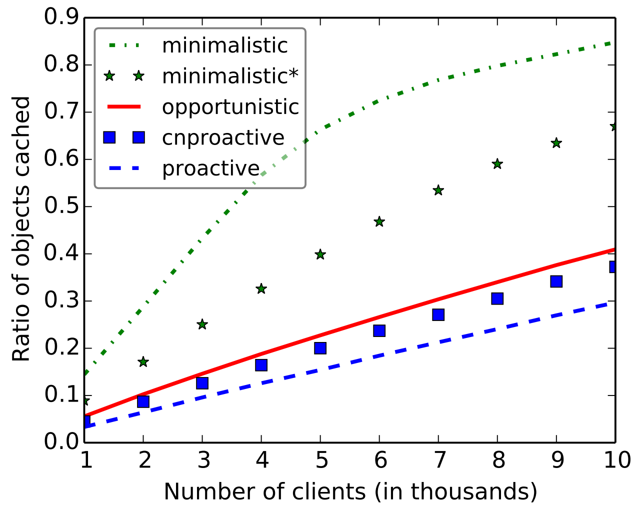
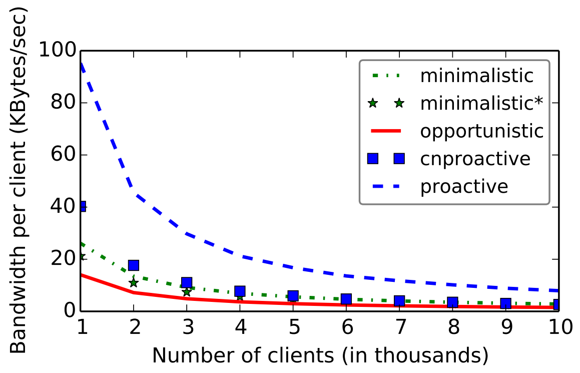
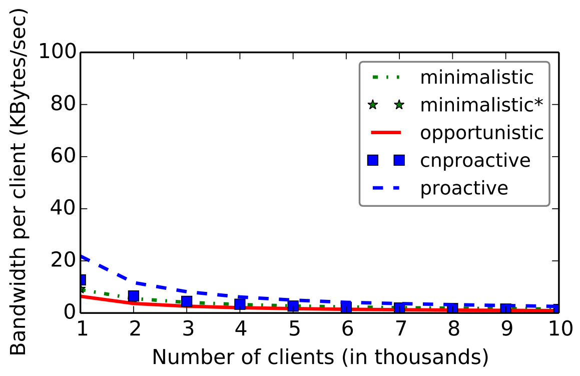


Fig. 5: Ratio of objects cached at the end of each simulation over the full set of objects requested for a varying number of clients.



(a) Outgoing bandwidth



(b) Outgoing bandwidth with churn

Fig. 6: Average outgoing bandwidth per client (a) and outgoing bandwidth running under a churn rate of 0.1 (b) for a varying number of clients.

5.2.3 *Client bandwidth*

Figure 6a shows average outgoing bandwidth per client as a function of the number of clients. Local outgoing bandwidth consists of two components: side-loading and pushing. The client bandwidth cost is proportional to the sum of those components. Naturally, a higher local hit ratio lowers fre-

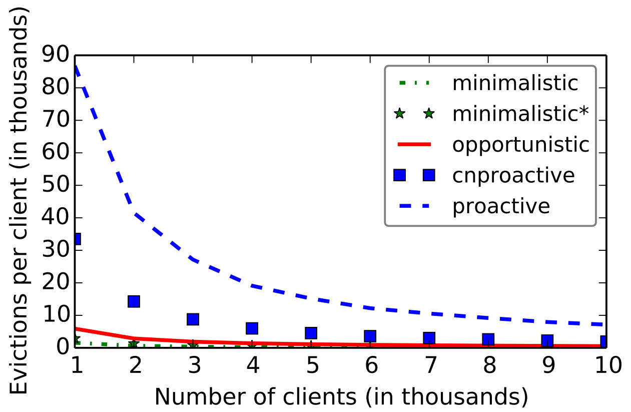


Fig. 7: Average number of evictions per client for a varying number of clients.

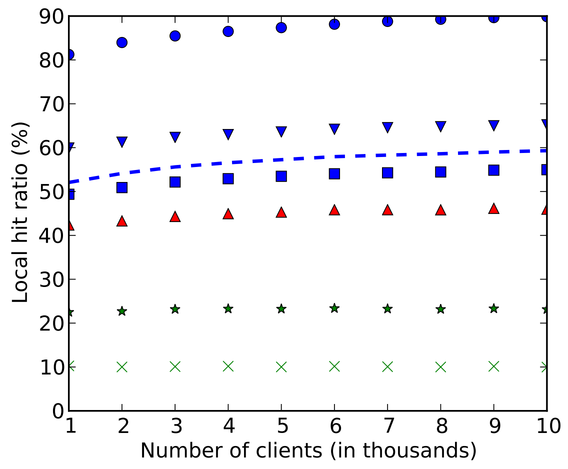
quency of side-loads and pushing (objects are only pushed to caches that do not contain the object). With that in mind it is easier to see why the opportunistic approach performs best. It performs no pushing while it keeps the number of side-loads low by creating a number of cache replicas that increases with demand. The minimalistic and minimalistic* approaches cannot perform as well with regards to client bandwidth, because they create fewer replicas and thus incur more side-loads.

The proactive approaches push content, which, as can be observed from Figure 6a, is more expensive than just side-loading. The reason for that is twofold: First, for each side-load, the proactive algorithms will additionally spend bandwidth pushing to a number of clients (the proactive approach more than the common neighbors variation). Although this results in increased local hit ratio, which saves bandwidth, it also results in a higher eviction rate.[1] This can be seen in Figure 7, which shows that the proactive approaches aggressively evict objects from the caches. Second, more evictions will result in more side-loads and thus more pushes as well. Consequently, the proactive approaches spend considerable bandwidth refilling the caches.

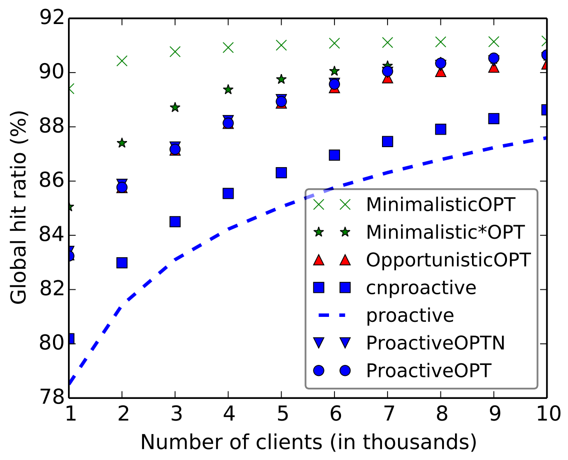
The common neighbors proactive approach has comparable performance as the opportunistic one. It achieves this by limiting the number of replicas it creates per side-load to a typically small fraction of the neighborhood (common neighbors only). At the same time it maintains a substantially higher local hit ratio per client, which saves additional bandwidth from future requests served locally on the requesting client.

With all caching approaches, the average bandwidth for side-loading and pushing decreases as the number of clients grows. This is because more clients imply a larger capacity of the collective cache. As the fraction of the corpus that can be stored in the cache increases, the number of evictions decreases (Figure 5). As a result, less side-loading as well as less pushing is needed.

1. Here we refer only to evictions that happen when a client's cache is full and a new object needs to be cached. Evictions initiated by the service are not considered. Unless stated otherwise, evictions refer to those that happen due to the cache replacement policy.



(a) Local hit ratio



(b) Global hit ratio

Fig. 8: Local hit ratio per client (a) and global hit ratio of the collective cache (b) of optimal algorithms as the number of clients increases.

5.3 Optimal case comparisons

In this section, we consider results of the optimal versions of the previously examined algorithms. This set of experiments identifies the best performance possible on the metrics discussed. Figure 8 depicts the results of these simulations. The optimal version of each algorithm is denoted by the name of the algorithm followed by the "OPT" suffix. Note that for the reactive approaches (opportunistic, minimalistic, and minimalistic*) their optimal counterparts are only optimal with respect to their eviction policy. The proactive approaches are also optimal with respect to their pushing strategies—they push content to exactly those clients that will be requesting this content in the future. We denote by "ProactiveOPTN" the algorithm that performs optimal pushing for each object within the neighborhood of the owner. We include the simple proactive and common neighbors proactive approaches for comparison purposes.

Figures 8 and 9 expose the potential of the proactive

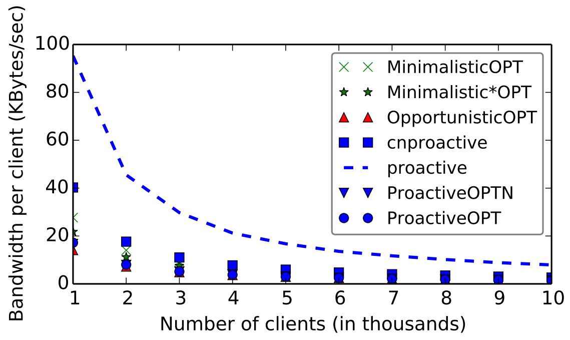


Fig. 9: Average outgoing bandwidth per client for optimal algorithms as the number of clients increases.

approaches for high local hit ratios, global hit ratios, and low client bandwidth overheads. While the reactive approaches are only marginally improved by an optimal eviction policy, the proactive schemes get a substantial boost on all metrics by pushing content optimally.

The optimal proactive approach (ProactiveOPT) achieves 90% local hit ratio for $10,000$ clients and perfect workload knowledge. If, however, we contain the optimal pushing for each object within the neighborhood of its owner (ProactiveOPTN), then the situation is drastically different. In fact, our non-optimal proactive approach and common neighbors proactive approaches approximate the ProactiveOPTN approach relatively well. The differences are within $7-10\%$. This is promising because confining ourselves to the neighborhood of the content owner is a reasonable choice given our target workload. Additionally, attempting to predict requests by clients outside of the owner's neighborhood may be harder to achieve reliably and is less scalable.

5.4 Replication factor

We now investigate the effect of the replication factor (proportion of the neighborhood that caches an object) on the proactive approach. We run a set of simulations with $10,000$ clients and vary the replication factor between 0.1 and 0.9. The results of this experiment can be seen in Figure 10. We omit a replication factor of 0, because in this case the algorithm becomes the opportunistic approach (no pushing takes place); the results for a replication factor of 1 have been presented in Figures 2a, 4a and 6a.

The findings show that as the replication factor increases both local hit ratio and average client bandwidth increase. This is because the number of replicas created increases with the replication factor (more pushing occurs) and thus more clients are likely to find the content of their neighbors in their caches (Figure 10a), thereby increasing the local hit ratio. The additional client bandwidth required by a higher replication factor is fairly small. The global hit ratio drops with the increase of the replication factor because the increasing number of object replicas take more space in the collective cache, leaving less "room" for additional objects to be cached. The decrease in global hit ratio is on the order of 1%, small compared to the corresponding increase in local hit ratio ($\sim$13%).

5.5 Churn

In Figures 2, 4 and 6, we present the results from a set of experiments where there is non-negligible churn in the system. We model churn as the fraction of clients leaving the system within a period of time. A client is considered to have left the system if the last time it has issued a request was before a predetermined session timeout period. We simulate a variety of churn rates by assigning appropriate values to the session timeout period.

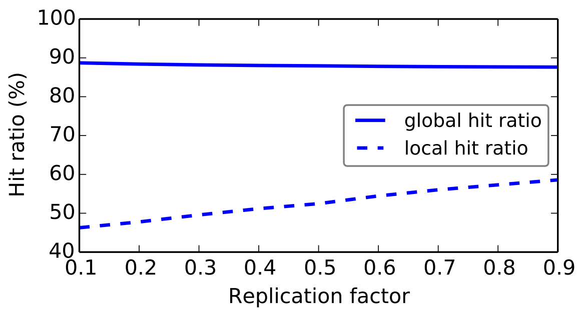
In Figure 4b we depict the results of the simulations for a churn rate of 0.1. The global hit ratio in Figure 4b has decreased substantially for all algorithms compared to the results without churn shown in the same figure (Figure 4). This is expected because increased churn means increased chance that a client goes offline soon after it caches an object. This in turn means that a subsequent request for the object will more likely be served by the service. As mentioned in Section 5.2.2, the minimalistic approach is the most sensitive to churn due to the low number of replicas it generates. The more replicas created by an approach, the less vulnerable it is to churn with respect to the global hit ratio. This is why the proactive approach outperforms the other schemes. Consequently, the global hit ratio of the algorithms is strongly correlated with the number of replicas per key they create (Figure 3b).

The results regarding clients' outgoing bandwidth in Figure 6b are similar to those of the base case (Figure 6a). The main difference is in the scale of the bandwidth ranges, which have substantially shrunk. This is because a large number of requests are now served by the service and thus clients spend less bandwidth serving content. (Recall that the service bandwidth is not accounted for in the outgoing client bandwidth metric.)

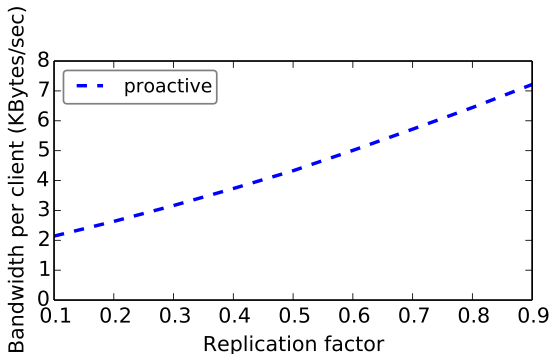
The more surprising results are those regarding the local hit ratio in Figure 2b. Observe that both proactive approaches are negatively affected by churn (Figure 2a). This is because the rate of clients going offline reduces the number of replicas they can create. This leads to a lower likelihood of a client finding content in its own cache.

The opportunistic approach, on the other hand, is not affected by churn, giving identical results as when clients remain online (Figure 2a). The reason is twofold: (i) In our implementation the cache of each client is persistent throughout the multiple sessions the client may open during each simulation run. This is a reasonable assumption given that we target client caches built on their browsers' web storage, which is persistent across sessions. (ii) The opportunistic scheme creates a replica of the requested object on the requesting client whenever a local cache miss occurs. The requesting client is always considered online because it opens or continues a session by virtue of issuing a request. Consequently, churn does not affect the number of replicas it can create and the local hit ratio remains the same.

The minimalistic and minimalistic* approaches benefit from churn with respect to the local hit ratio metric. If a client caching an object goes offline, the next request for the object will generate a new cache copy (from the service's perspective, no client exists that caches the data). When the client that

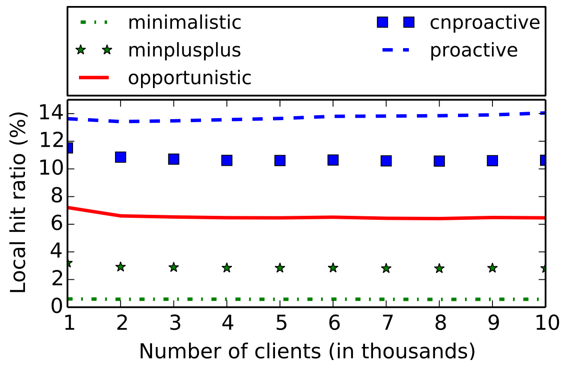


(a) Hit ratio

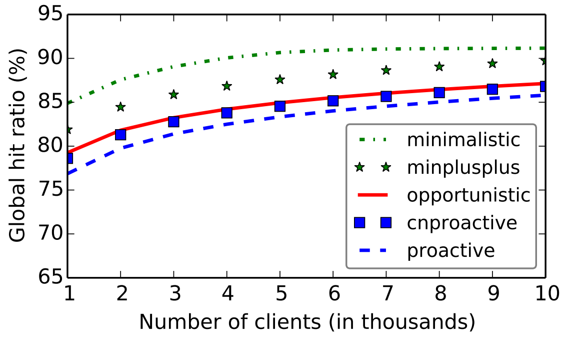


(b) Local outgoing bandwidth

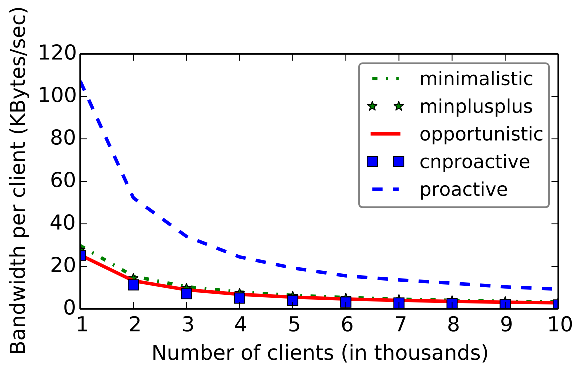
Fig. 10: Average local hit ratio per client and global hit ratio of the collective cache (a) and average bandwidth per client (b) for the proactive approach running with 10,000 clients as the replication factor increases.



(a) Local hit ratio



(b) Global hit ratio



(c) Outgoing bandwidth

Fig. 11: Average local hit ratio (a), global hit ratio (b) and average outgoing bandwidth (c) per client running under *NAP* $= 0.2$ for a varying number of clients.

caches this object comes online and starts a new session, two copies will be available in the collective cache because caches carry over different sessions in the same simulation run. Consequently, the minimalistic approaches can generate more replicas per key under churn, which results in an increase of local hit ratio.

We observed similar results for higher levels of churn rate, though the differences were more pronounced. Minimalistic and minimalistic* did even worse with respect to global and local hit ratio. In addition, the gap between proactive and opportunistic approaches increased for the global hit ratio and decreased for the local hit ratio. The results regarding bandwidth were similar to those in Figure 6b with decreased bandwidth spent in pushing and side-loading.

5.6 Neighborhood Access Probability

So far we have investigated the behavior of client caches under workload characteristics that substantially skew object accesses within the neighborhood of the owner of the object. The magnitude of this skew is determined by *NAP*, which for the results presented so far was pinned to 0.8 as literature [8] suggests for OSNs. In this section, we observe the performance characteristics of the previous caching strategies for *NAP* $= 0.2$, that is, a small workload skew. This means that 80% of requests for an object come from randomly selected clients outside the object owner's neighborhood in the client relationship graph. Note that given the graph sizes and the average neighborhood sizes shown in Figure 3a, setting *NAP* to such a low value renders the requesting client being chosen almost uniformly at random, 80% of the time. The objects' popularity and sizes distributions remain the same and the read/write ratio is preserved.

We perform the same experiments as in Section 5.2. In Figure 11a, we observe how local hit ratio is affected by the size of the system. Local hit ratio has drastically decreased for all algorithms compared to Figure 2a, because objects are now accessed by a wider set of clients. The ordering of the protocols with respect to their local hit ratio is still as in Figure 2a because strategies that create more replicas have a better local hit ratio. For all algorithms the local hit ratio is not much affected by the number of clients.

Strategy	LHR	LHR (Churn)	LHR (OPT)	GHR	GHR (Churn)	GHR (OPT)	B/W	B/W (Churn)	B/W (OPT)
Minimalistic	9.98	22.54	9.97	90.62	65.40	91.17	2.81	1.51	2.83
Minimalistic*	23.03	30.00	23.14	89.79	68.66	90.60	2.31	1.35	2.33
Opportunistic	45.02	45.02	45.97	89.11	72.15	90.31	1.51	0.92	1.51
Common Neighbors	54.97	46.59	65.21	88.62	73.98	90.54	2.64	1.33	1.49
Proactive	59.31	47.90	89.86	87.59	74.93	90.65	7.92	2.53	1.46

TABLE 2: Local hit ratio (LHR), global hit ratio (GHR) and average client bandwidth in KB/s (B/W) for each strategy with and without churn and optimal variants for 10,000 clients. The optimal variants of common neighbors and basic proactive approaches correspond to ProactiveOPTN and ProactiveOPT (from Figures 8a, 8b, and 9) respectively.

Strategy	# replicas / object	# evictions / client	% objects cached (%)
Minimalistic	1.0	29	84.80
Minimalistic*	1.94	132	67.01
Opportunistic	3.52	536	40.95
Common Neighbors	3.87	1926	37.23
Proactive	4.83	7140	29.67

TABLE 3: Average number of replicas per objects, evictions per client, and ratio of objects cached for each caching strategy for 10,000 clients.

Figure 11b shows results for the global hit ratio. The results are similar to our prior experiments (Figure 4a), the main difference being that all approaches show slight decrease in global hit ratio. This is because object accesses are more spread over the set of clients, resulting in more replicas per object and less cache capacity for storing additional objects.

The average outgoing bandwidth shown in Figure 11c is also similar to previous experiments (Figure 6a). Two points are worth noting: First, all approaches spend more bandwidth because access is more spread out and more side-loads as well as pushing occur. Second, the common neighbors proactive approach performs similar to the opportunistic approach because it is less likely that there are common neighbors between the requesting client and the owner of the requested object for small *NAP*.

5.7 Results Summary

Our simulations demonstrate how we can employ workload-specific information to improve performance and efficiency of cooperative caching. With small overhead on service load ($\sim$2%) and cost in client bandwidth, we can substantially improve client perceived latency (Figures 2a, 4a, and 6a). The relative costs of proactive replica creation and placement decrease as the system size increases. This suggests that our approaches are promising for systems of larger scale. By controlling the amount of proactivity in our strategies (Figures 10a and 10b), we can drive these costs to be competitive to the widely used opportunistic approach while retaining most of the benefits of the proactive approach. In addition, our proactive approaches approximate the optimal cache placement strategy on local hit ratio within a relatively small range ($\sim$7%) when the optimal placement is confined within the neighborhood of the owner of the requested object (Figure 8a). Our proactive approaches handle churn more gracefully (Figures 2b, 4b and 6b) because of the increased number of replicas created per object (Figure 3b). Finally, our approaches remain competitive even under workloads with lower locality (Figure 11). Tables 2 and 3 summarize the previous results for 10,000 clients and $NAP = 0.8$.

6 Related Work

Cooperative caching is a well-studied topic, particularly in the context of web caching [19], [20]. A representative system in this setting is the Shark [21] distributed file system, which uses sloppy DHTs to build locality-aware cooperative P2P caches between proxies. Another such system is Backslash [3], a collaborative P2P server/proxy caching for dealing with flash crowds using request redirection and URL rewriting. These systems focus on cooperative caches on proxy servers, hence they do not have the restrictions and dynamics of caches built on clients.

Our system model is based on the Maygh [2] cooperative client-side caching system. Maygh tries to reduce load on a web service by building a Content Delivery Network (CDN) on the clients' browsers. Another such system is Squirrel [1], a P2P web caching mechanism for geographically collocated clients (for example, within the same company) that uses a DHT for achieving scalability, self-organization, and churn tolerance. Our work focuses on OSNs, which are not a good candidate for Squirrel due to its requirement for geographically collocated clients. Also, neither of these approaches considered the problem of cache placement.

The cache placement problem has been studied in various settings. Several placement algorithms for web server replicas have been developed and evaluated in [22]. They use workload information, such as client latency and request rates, to make informed placement decisions. Our work performs cache placement by employing social link information and client preferences instead.

The work in [23] studies cache placement for web proxies under the assumption that the underlying network has a tree topology, modeling cache placement as a dynamic programming problem. [24] focuses on minimizing load on the network assuming a tree structure of limited depth and formulating a local greedy approach for finding a near-optimal solution. The work in [25] considers various replication and caching strategies within a simulated grid environment. The work views the grid as a tiered system and uses dynamic replication strategies to improve data access. Due to their restricted topology, these approaches do not generalize to clients of an OSN.

[26] proposes a cache placement strategy in OSNs similar to ours. This approach, called S-Clone, collocates replicas of data of neighboring clients with respect to the social graph of an OSN. Unlike our work, S-Clone replicates content on servers and not clients, avoiding the limitations of small cache capacities and churn. Also, S-Clone does not take workload information into account.

7 Conclusions

In this work we studied the cache placement problem for a cooperative cache built on the clients of an Online Social Network. We considered a service that maintains a directory of content locations and that has access to the social links between clients. The service provides the clients with hints about where content can be found as well as where it should be cached. Given this system model, we proposed proactive cache placement strategies that leverage social links between clients.

We evaluated the proposed proactive schemes using simulations on synthetically generated graphs and workloads whose characteristics match those of real OSNs and compared them with various baseline approaches. Our findings show that proactively caching content where it is most likely to be requested can achieve substantial improvements in hit ratio over commonly used approaches at moderate overheads.

We also implemented optimal versions of the various cache placement strategies we considered. These approaches know the entire workload ahead of time. In one of our proactive variants, the hit ratio is only $\sim$7% lower than optimal.

Finally, we explored the effects of churn and showed that our proposed proactive approaches are effective even under moderate churn.

Acknowledgments

We are grateful for the anonymous reviews and the partial funding by grants from DARPA, NSF, Facebook, and Microsoft Corporation.

References

[1] 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)*, 2002, pp. 213–222.

[2] 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)*, 2013, pp. 281–294.

[3] 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)*, 2002, pp. 203–213.

[4] S. Nikolaou, R. van Renesse, and N. Schiper, “Cooperative client caching strategies for social and web applications,” in *Large-Scale Distributed Systems and Middleware (LADIS’13)*, November 2013.

[5] L. Ramaswamy, L. Liu, and A. Iyengar, “Cache clouds: Cooperative caching of dynamic documents in edge networks,” in *25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, 2005, pp. 229–238.

[6] 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)*, 2012, pp. 53–64.

[7] 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 World wide web (WWW’10)*, 2010, pp. 861–870.

[8] 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)*, 2009, pp. 49–62.

[9] 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)*, 2003, pp. 115–130.

[10] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, Feb. 1978, pp. 120–126.

[11] L. A. Belady, “A study of replacement algorithms for virtual storage,” *IBM System Journal*, vol. 5, 1966, pp. 78 – 101.

[12] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, “An analysis of facebook photo caching,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP ’13)*, 2013, pp. 167–181.

[13] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in *Proceedings of the Eleventh*

ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05), 2005, pp. 177–187.

[14] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Phys. Rev. E*, vol. 64, no. 2, Jul 2001, p. 026118.

[15] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014 [2012].

[17] "HTML5 local storage," http://www.w3schools.com/html/html5_webstorage.asp, 2015 [2014].

[18] H. Geng and R. Renesse, "Sprinkler—reliable broadcast for geographically dispersed datacenters," in *Middleware 2013*, ser. Lecture Notes in Computer Science, D. Eyers and K. Schwan, Eds. Springer Berlin Heidelberg, 2013, vol. 8275, pp. 247–266.

[19] 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)*, 1999, pp. 16–31.

[20] J. Wang, "A survey of web caching schemes for the Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, Oct. 1999, pp. 36–46.

[21] 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*, 2005, pp. 129–142.

[22] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of web server replicas," in *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '01)*, vol. 3, 2001, pp. 1587–1596.

[23] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby, "On the optimal placement of web proxies in the Internet," in *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM'99)*, vol. 3, 1999, pp. 1282–1290.

[24] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *Proceedings of the 29th Conference on Information Communications (INFOCOM'10)*, 2010, pp. 1478–1486.

[25] K. Ranganathan and I. T. Foster, "Identifying dynamic replication strategies for a high-performance data grid," in *Proceedings of the Second International Workshop on Grid Computing (GRID'01)*, 2001, pp. 75–86.

[26] D. A. Tran, K. Nguyen, and C. Pham, "S-clone: Socially-aware data replication for social networks," *Computer Networks*, vol. 56, no. 7, 2012, pp. 2001 – 2013.

Stavros Nikolaou is a fourth year Ph.D. student in the Department of Computer Science at Cornell University. His research interests lie in distributed systems and more specifically in their scalability and self-organization. He is also interested in algorithms, especially those designed for distributed computation.

Robbert van Renesse is a Principal Research Scientist in the Department of Computer Science at Cornell University. He received a Ph.D. from the Vrije Universiteit in Amsterdam in 1989. After working at AT&T Bell Labs in Murray Hill he joined Cornell in 1991. He was associate editor of IEEE Transactions on Parallel and Distributed Systems from 1997 to 1999, and he is currently associate editor for ACM Computing Surveys. His research interests include the fault tolerance and scalability of distributed systems. Van Renesse is an ACM Fellow.

Nicolas Schiper obtained his Ph.D. degree in 2009 from the University of Lugano, Switzerland. He spent slightly more than two years as a postdoc at Cornell University, doing research on scalability and energy efficiency aspects of distributed systems. He recently joined industry where he works on facilitating the integration of heterogeneous systems.