Behave: Behavioral Cache for Web Content

Davide Frey, Mathieu Goessens, and Anne-Marie Kermarrec

INRIA-Rennes Bretagne Atlantique, Rennes, France

Abstract. We propose Behave: a novel approach for peer-to-peer cache-oriented applications such as CDNs. Behave relies on the principle of Behavioral Locality inspired from collaborative filtering. Users that have visited similar websites in the past will have local caches that provide interesting content for one another.

Behave exploits epidemic protocols to build overlapping communities of peers with similar interests. Peers in the same one-hop community federate their cache indexes in a Behavioral cache. Extensive simulations on a real data trace show that Behave can provide zero-hop lookup latency for about 50% of the content available in a DHT-based CDN.

1 Introduction

Publishing content on the Internet has become a daily task for a large number of users. People publish content on social networks, blogs, or on their own websites on a daily basis. The increasing availability of broadband connectivity has even prompted users to start hosting their websites themselves on small servers that are always on in their homes. Small inexpensive plug computers like the Raspberry Pi [1] allow users to set up a personal web server for as little as $30. Software solutions like FreedomBox [2] allow even inexperienced users to deploy their own websites on their plug computers within minutes.

Traffic demands for most of these self-hosted websites may appear negligible. But experience has shown that even small websites can experience instant surges of traffic due to the accidental discovery of interesting content [3]. This phenomenon, known as flash-crowd, often results from traffic redirection from social media or other popular websites. Yet, even with a very limited number of visitors, websites hosting relatively heavy content like high-definition pictures or videos can easily saturate the upload capacity available to residential users.

Commercial websites address traffic surges by relying on the elastic capabilities of Content Delivery Networks (CDN), which delegate content distribution to servers located as close as possible to users. By specializing on content delivery, CDNs platforms such as Akamai [4] allow client websites to respond to sudden traffic demands in real time without maintaining costly infrastructures.

Unfortunately, while affordable and definitely convenient for large corporate websites, CDNs remain too expensive for personal websites or even for those of small- and medium-sized businesses. Researchers have tried to address this disparity by exploiting the bandwidth and storage capabilities of users that visit a website to provide the same content to other users. This results in peer-to-peer [5–7], or peer-assisted [8] solutions that aim to provide a cheaper and thus widely accessible alternative to CDNs. However, their success is limited by the latency they introduce in the lookup of content.

Most peer-to-peer or peer-assisted CDNs rely on Distributed Hash Tables (DHTs). This provides a simple abstraction that allows nodes to retrieve the content they are looking for in a logarithmic number of hops. However, even a logarithmic number of hops may be too long, particularly in the presence of congestion, or churn. DHTs may suffer from long lookup delays in the order of minutes [9, 10] in the presence of churn. But even in a static network, typical DHT lookup times easily exceed the download times required by most content. Some authors have proposed to improve on this latency with new DHT solutions [9, 11]. Here we adopt a complementary approach and reduce lookup time to zero by maximizing the amount of content that nodes can index locally.

Our key contribution consists of *Behave*, a novel decentralized caching architecture relying on *behavioral locality*. A traditional CDN essentially consists of a large decentralized cache memory, relying on temporal locality. If a user accesses a web object, there is a relatively strong probability that she will access it again in the near future. Behave's *behavioral locality* extends this observation by exploiting similarities between the browsing behaviors of users. Unlike DHT-based solutions [6, 5], which seek to maximize the amount of web objects accessible through the peer-to-peer substrate, Behave focuses on maximizing the number of objects accessible without any lookup delay.

To achieve this, Behave builds a collaborative *behavioral cache* by relying on user-based collaborative filtering [12], a well-known technology in the context of recommender systems. Collaborative filtering relies on a simple observation: users that have exhibited similar tastes in the past will probably exhibit similar tastes in the future. In our context, this observation suggests that if two users have common items in their web caches, then they will probably exhibit commonalities in the websites they visit in the future.

We adopt a decentralized version of user-based collaborative filtering following the example of [13]. Behave nodes adopt an epidemic protocol to form an interest-based topology based on commonalities between their browsing histories. This provides each node with a set of overlay neighbors whose browsing histories most closely resemble its own. Behave's *behavioral cache* emerges from this topology as the federation of the local caches of a node's neighbors.

Behave provides the greatest benefit for applications for which lookup time using a DHT may exceed the time needed to effectively fetch the content itself. The web provides a good example of such applications, together with applications such as decentralized Domain Name Services [14]. While web pages do contain large objects, most of their content consists of a large number of relatively small files. For example, in a sample of 300,000 web requests from the top 1,000,000 websites crawled by the HTTP Archive initiative [15], we recorded an average web-page size of 1.62MB consisting of an average of 94 files per page, with an average size of 18KB each. The average lookup time on a DHT may significantly exceed the time required to download most of these files. Our extensive simulations on real data traces show instead that Behave can provide zero-hop lookup latency on 50% of the content indexed by an entire DHT. Moreover, mimicking the performance of a single Behave overlay with existing solutions such as FlowerCDN requires each node to participate in more than 10 gossip overlays.

2 The Behave Model

Behave maximizes the amount of cached information reachable without recurring to on-demand routing operations. To this end, it combines the notion of temporal locality at the basis of standard web-caches with a novel idea: behavioral locality. Two users that visited the same websites in the past will likely exhibit more common interests in the future.

From a practical perspective, Behave extends the caching behavior commonly implemented in web browsers by integrating a decentralized user-based collaborative filtering protocol [12, 16, 17]. Each node identifies the set of nodes that are most similar to it in terms of browsing history: its neighbors. The aggregation of the cache indexes of a node's neighbors constitutes a *behavioral cache index*. If a node visits a website not indexed by its local cache, the site will likely be in the node's behavioral cache index. If this is the case, the node will download the content directly from the corresponding neighbor with no lookup latency.

Like most peer-to-peer solutions for web-content delivery, Behave relies on signatures to ensure the integrity of content retrieved from other nodes [18, 19]. Yet, for the sake of simplicity, we ignore privacy-preserving solutions for Behave-like systems such as those in [13, 20]. Integrating them in Behave constitutes part of our future work.

Implementation. We implemented a preliminary Behave prototype in the form of a local proxy server consisting of 4000 lines of Java code. We developed a web proxy using Apache HTTP Components (`http://hc.apache.org`), and implemented a cache conforming to the HTTP/1.1 caching policies. We manage the cache using Ehcache (`http://ehcache.org`), and profiles with a small-footprint SQL Java database (`http://www.h2database.com`). Users can access Behave with any browser simply by configuring it to use *localhost:8080* as a proxy server. This allows our implementation to handle all requests and serve them either using the behavioral cache, or by directly contacting the target server.

We are also working on a Firefox-based implementation that leverages WebRTC, as well as the browser's internal API (Mozilla's XPCOM). Despite being in an early design stage, our code is already available at `https://github.com/mgoessen/behave/`. In the remainder of this section, we detail the concepts underlying the Behave architecture.

2.1 From Local Cache to Interest Profile

Like any web caching solution, Behave relies on the notion of a *local cache* based on the principle of temporal locality. If a user visits a website, there is a non-negligible probability that she will visit it again in the future. In this paper, we complement this basic idea with *behavioral locality*. For the sake of simplicity, we consider an LRU-based local cache that associates each URL visited by the local user with the corresponding web object. We use the term *local cache* to refer to the stored URLs and the associated web objects. We write instead *local-cache index* to refer to the list of URLs without the web objects.

Compact Representation of Cache Indexes. The local cache allows each node to retrieve copies of the websites it visited in the past. Behave makes these copies also available to other nodes that have similar browsing histories. To achieve this, nodes share the content of their cache indexes with other similar nodes. This poses an evident problem. Even if the local cache index does not contain the actual web objects, its size may still be very large. Considering the average length of a URL—we measured an average URL length of 99 characters over a set of 300,000 web pages—the index of a cache index containing 2000 URLs takes about 20KB. This grows even larger when we append HTTP content-negotiation headers [21]. Blindly exchanging cache indexes without any form of compression would therefore result in prohibitive network overhead.

To counter this problem, Behave uses a compact representation of local cache indexes in the form of bloom filters [22]. A bloom filter represents a set as an array of m bits by exploiting k hash functions. Each such function returns a number from 0 to $m-1$ when applied to an item (a URL in our case). Adding an element (URL) to the set is achieved by hashing it with all the k hash functions and by setting to 1 all the bits corresponding to the values returned by each of the functions. Checking if an element is in a set relies on an analogous process: hashing the element with the k functions and verifying if all the resulting k bit positions are set. If they are, the element is deemed to be in the set, otherwise it is not. It is easy to see that a Bloom filter can return false positives, but can never return false negatives.

Behave determines the size of bloom filters dynamically. Periodically, a Behave node recomputes its own bloom filter with a size of $b \cdot N$ where N is the cardinality of its profile or cache index. Such a periodic recomputation not only achieves similar size efficiency as scalable bloom filters [23], but it also allows a bloom filter to ignore the items that have been evicted from the corresponding node's local cache. When sharing its bloom filter, a Behave node appends the corresponding size so that receiving nodes can parameterize the hash functions accordingly [24]. We evaluate the impact of b in Section 4.5.

Interest Profile. The compressed form of a local-cache index allows a node to inform another node of its browsing interests. However, not all the items accessed through a web browser appear in the corresponding local cache and thus in the local-cache index. The *HTTP* specification allows [21] web designers to identify objects as *cachable* (with a specified TTL and refresh policy, allowing even dynamic content to be cached) or *non cachable*. Web browser caches only store cachable objects and Behave's local cache does the same. Not respecting this specification would clearly result in the undesirable use and propagation of stale copies of documents.

To improve the accuracy of their browsing histories while still respecting the specification, Behave nodes therefore complement the information contained in their local cache indexes. Specifically, each node maintains a separate *interest profile* consisting of a list of visited URLs. All the items in the local cache and local-cache index have a corresponding entry in the interest profile, but non-cachable items appear in the interest profile without being present in the local cache. This allows nodes to gather more information about the browsing histories of potential neighbors than would be available in their local cache indexes.

Like for the local cache index, we use a compact representation of the interest profile of a node in the form of a bloom filter, which is periodically recomputed with an appropriate size. However, we must point out an important difference between the two. Nodes use interest profiles to identify similarities between their browsing behavior and that of other nodes, but they use cache indexes to verify if a given web object is really stored locally or in the cache of a neighbor. Thus, profiles can tolerate much higher rates of false positives. A false positive in a node's profile will probably appear as a (false) positive in the profile of a neighboring node. A false positive in a cache index will instead lead to a cache miss possibly resulting in a waste of time and network resources. We will return to this important distinction in the context of our evaluation in Section 4.5.

2.2 Clustering Similar Interests

To cluster nodes according to interests, Behave adopts the approach in [13] consisting of two layered gossip protocols: random peer sampling, and clustering. The former provides each node with a continuously changing random view of the network. The latter starts from this random view, and incrementally identifies the best neighbors for each node according to a given similarity metric.

The two protocols follow similar structures. Each node maintains a *view*—a list of identifiers (for example IP address and port) of n_{view} other nodes, each associated with the corresponding interest profile, and with a timestamp indicating when the information in the entry was generated. Periodically, each node selects the entry in its view with the oldest timestamp and starts a gossip exchange with the corresponding node. Consider a node p starting such an exchange with another node q. Node p extracts a new view-like data structure G from its view: a random subset of $n_{view}/2$ entries for the RPS, and a copy of its view for the clustering protocol—in both cases node p excludes q from the extracted set. After preparing G, node p sends it to q thereby initiating the gossip interaction. The two protocols differ in the way q reacts to G.

Random Peer Sampling. In the case of the RPS, q selects $n_{view}/2$ entries from its own view and replaces them with those in G. It then takes the entries it removed from its view and packs them into a response message G', and sends them to p. Upon receiving G', node p replaces the entries it sent to q with those it received, thus completing the gossip interaction.

Clustering Protocol. In the case of the clustering protocol, q computes the union of its own cluster view, its own RPS view, and the view in G (which is p's cluster view). Then q selects the nodes in the resulting view whose profiles are most similar to its own. Several similarity metrics exist for this purpose. Here we use the one proposed in [13]. It extends the cosine-similarity metric by providing ratings for groups of nodes as opposed to individual ones. This allows q to identify the nodes that collectively best cover the interests in its own profile. After selecting the nodes to keep in its view, node q prepares a reply G' with the content of its own view before the update and sends it to p, which reacts by updating its view in an analogous manner.

Nodes open connections to other nodes using state-of-the-art mechanisms such as IPv6, ICE [25], or UPNP/PCT [26], and maintain these connections stable with the nodes in their clustering views. This speeds up the download of content as nodes do not need to wait for connection initialization. Moreover, it allows them to detect and quickly respond to other nodes' disconnections.

2.3 Collaborative Cache Index

Identifying neighbors with similar interest profiles allows Behave nodes to build collaborative *behavioral cache indexes* that federate the local cache indexes of their neighbors. In practical terms, the *behavioral cache index* of a node simply consists of a data structure that associates each of the node's neighbors with the corresponding local cache index in its bloom-filtered form. The use of bloom filters prevents nodes from organizing their collaborative cache indexes in a more efficient structure. Yet, the small number of neighbors makes the search for matching bloom filters sufficiently fast.

2.4 Speeding Up Convergence

To maximize the efficiency of behavioral cache indexes, nodes must maintain them up to date. Moreover, they must base their clustering decisions on up-to-date versions of other nodes' profiles. The gossip exchanges described above periodically refresh the information about other nodes' profiles. But they only do so to a limited extent. At each gossip cycle, the RPS exchanges information with a randomly selected node and thus cannot provide significant help in maintaining the information about one's neighbors current. The clustering protocol instead exchanges information exactly with one of the node's neighbors. Receiving updated information about n_{view} nodes therefore requires on average n_{view} gossip cycles. This is too long to provide satisfactory performance.

To maintain their behavioral cache up-to-date, nodes therefore complement their gossip exchanges by explicitly pulling profile information. At each gossip cycle, a node sends an explicit *profile request* message to each of its current neighbors from its cluster view. Nodes reply to such *profile requests* by sending copies of their interest profiles and local cache indexes.

To limit the cost of such profile updates, nodes do not reply systematically, but they use a profile-change threshold, t, expressed as a percentage of the size of their interest profile. When a node q replies to a node n, it records a snapshot of the profile it sends to n. When q receives a subsequent request from n, it first checks if its current profile contains at least t new elements with respect to the one it sent the last time it replied to a request from n. If so, q replies by sending its new profile and its updated cache index, otherwise it ignores the request.

It is worth observing that this threshold-based approach integrates particularly well with bloom filters. Alternatives like differential updates would be inapplicable due to the difficulty of removing items from a bloom filter. Moreover, differential updates with lists of full URLs would often be larger than non-differential bloom filters.

Finally, we observe that using profile requests is not equivalent to increasing the frequency of gossip interactions. In gossip interactions, a node exchanges profile information about n_{view} other nodes, while each profile request results in the exchange of at most one profile.

3 Evaluation Setting

We evaluated Behave on two real-world traces provided by the Ircache-DITL[1] initiative—a web proxy farm with servers deployed at various US universities and research institutes. The first trace refers to January 9, 2007, the second to January 10 of the same year. Each records 24 hours of logs for approximately 1000 clients. After removing NAT endpoints, the traces contain, respectively, $4.2M$ and $3.8M$ requests from 982 and 1000 unique clients for about $2M$ and $1.7M$ web objects with average sizes of $24.3KB$ and $23.7KB$. Ircache also provides more recent datasets, but these only contain data for 150–200 clients. Due to the absence of caching information in the dataset, we assume that 46% the content is cachable, consistently with the data published by the HTTP Archive initiative [15]. Similarly, we assume that all clients negotiate content in the same language. Even if the dataset does not contain such information, this is probably close to reality as the dataset records web accesses from US Universities. Finally, we assume that every client remains online during the entire experiment.

Default Parameters. Due to the large size of the datasets, we implemented a simulated version of Behave, which we configured as follows. We set the sizes of the RPS and clustering views respectively to $v_{\mathrm{RPS}} = 12$ and $v_{\mathrm{CLUSTER}} = 25$. We used a gossip period $p_g = 10\text{min}$ for both protocols and a profile-update period $p_u = 1\text{min}$ with a threshold of $t = 5\%$. Finally, we set the profile size to 5000 elements and the cache size to 2300 with bloom filters that use respectively 20 and 12 bits per element. We provide details about the choice of these default parameters in sections 4.3 through 4.5.

Comparison with a DHT. We compare Behave with a Squirrel-like DHT configured to use a base $b = 4$, a leaf-set size $l = 16$, a cache size of 2300 elements like

[1] http://ircache.net National Science Foundation (grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research.

for Behave, and an average insertion latency of 5s. To manage the CDN data, we use the *home-store-node* strategy described in [6]: the node associated with a key in the DHT stores a pointer to the node that records the actual content.

Comparison with FlowerCDN. We also compare our solution with FlowerCDN [5]. FlowerCDN reduces the lookup latency for already-visited websites by employing per-site clusters. A node joins a cluster after it visits a website for the first time, and maintains a partial view for each of the clusters it is part of. The authors of [5] do not specify how many clusters a peer can join, so we experimented with different maximum numbers of clusters with a simple LRU policy.

If a node visits a website associated with one of its clusters, it can lookup the peer hosting the content within zero hops if the content is indexed in its partial view, or within one hop if the content is in the cluster but not in its partial view. For simplicity, we ignore localities which would decrease the cluster hit-rate, and assume that joining a cluster takes 1s. We also ignore the delay for the propagation of caches indexes within a clusters: content is immediately available after the 1s joining delay. As above, we set the local cache size to 2300 entries.

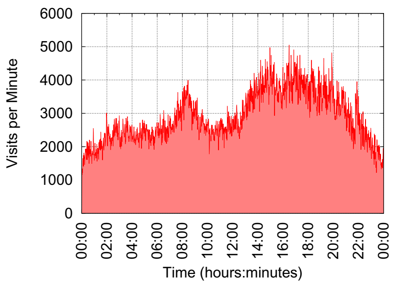
4 Evaluation Results

We start by comparing the performance of Behave with that of Squirrel and FlowerCDN. Then we analyze Behave by varying its main parameters one by one. When not otherwise specified, we use the default parameters indicated in Section 3. We also point out that our results showed no significant differences between the two traces we considered. Thus, we only present those for January 9, 2007. Figure 1a gives a pictorial view of this trace by showing the total number of visited objects per minute during the course of the day.

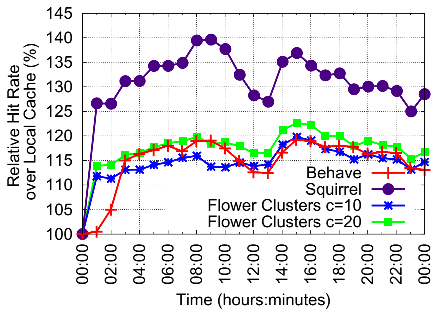
4.1 Comparison with Squirrel and FlowerCDN

Behave aims to provide zero-hop lookup latency on as large a fraction of the content as possible. DHT-based solutions, on the other hand, focus on indexing all the content available at all peers even if this would result in a long lookup latency. In spite of this significant difference, Behave manages to achieve zero-hop lookup latency for up to 50% of the content indexed by the DHT.

Figures 1b shows how the hit rate progresses during the course of the experiment. The plot (and the plots that follow) shows, for each instant, the relative hit rate with respect to a solution that uses only the local cache. Both Behave and its competitors leverage the local cache. Thus, a hit rate of 110% means that the solution offers 10% more content than the local cache alone. Each point represents the hit rate in the previous hour of simulated time. Clearly, the DHT-based solution (indicated as Squirrel) constantly achieves a higher hit rate, but Behave still manages to offer a large fraction of the content available through the DHT. Moreover, its Behavioral cache index provides this content with zero lookup latency. Squirrel, on the other hand, serves a pointer to the content, or notifies of its absence, with a significantly higher lookup delay.

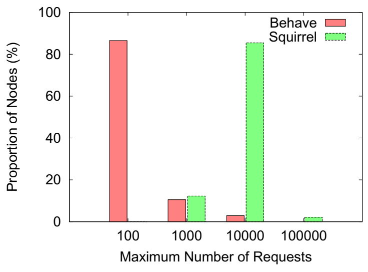


(a) Visits per minute.

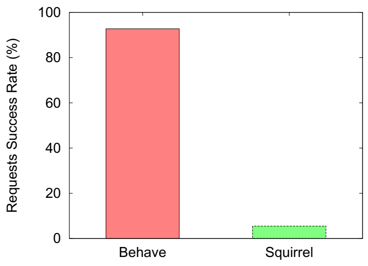


(b) Behave, Squirrel and FlowerHit Ratio.

Fig. 1. Behave, Squirrel and FlowerCDN Hit Ratio and Visits Pattern



(a) Query load Distribution.



(b) Request success rate.

Fig. 2. Query load as the number of requests received by nodes (a), and request success rate

The plot also shows the hit-rate obtained by the gossip-based overlay in FlowerCDN (labeled as Flower-clusters) for different maximum numbers of clusters. The data shows that Behave's hit-rate is between that achieved by FlowerCDN's gossip overlays with a maximum of 10 and 20 clusters. However, the cost associated with Behave's gossip exchanges roughly equals that of FlowerCDN with four clusters (Behave nodes exchange profiles and cache indexes both in the clustering and RPS protocols). This shows that a behavioral cache like Behave can be effective both as a complement and as an alternative to DHT-based solutions.

4.2 Query Load

Figure 2a presents another interesting aspect of our comparison with Squirrel by plotting the distribution of nodes according to the number of requests for content they receive (query load). The plot shows that most Behave nodes receive less than 100 requests with only a few nodes receiving more. The distribution of Squirrel nodes on the other hand is a lot more spread out. A large number of nodes receive thousands of requests: more than 80% receive more than 10000 requests (data for FlowerCDN would likely be similar). To explain this behavior, we observe that in DHT-based approaches, each piece of content is managed by one or a few nodes. With power-law content distributions such as the web, this inevitably overloads nodes that are responsible for very popular content. Behave, on the other hand, naturally replicates content according to its popularity.

Figure 2b highlights the different approach taken by Behave with respect to Squirrel and FlowerCDN. It plots the percentage of requests that can be satisfied by the requested peer, over the total number of requests it receives. Note that this is not the same as the hit/miss rate. Behave only requests content from a peer when the corresponding cache index indicates that the content will be available. So a cache miss in Behave does not generally result in a request. The only exceptions arise when a bloom filter appears to contain an item that is not in the corresponding cache. This may happen because of the filter's false positive rate, or because the item has been removed from the cache after the bloom filter was created. Both turn out to be rare occurrences thanks to the size of bloom filters (analyzed in Figure 5), to the short update period of 1 minute, and to the small update threshold of 5%.

Unlike Behave, Squirrel and FlowerCDN always try to retrieve content from the DHT. This means that they fall back to the origin server only after a failed routing attempt which requires $log(n)$ hops. This highlights Behave's ability to avoid useless communication, ultimately leading to reduced network overhead and higher responsiveness.

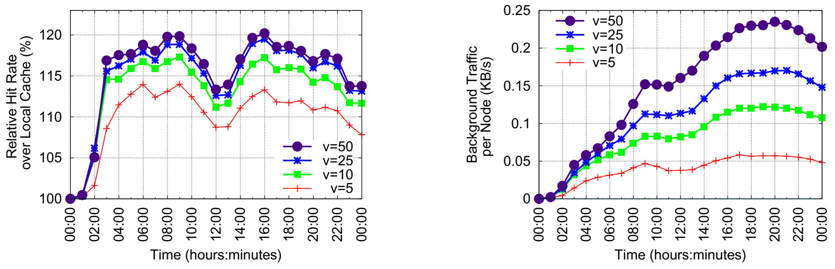
Behave's responsiveness also stands out in the case of satisfied requests. By construction, Behave honors all requests in 0 hops. On the other hand, Squirrel honors over 50% of requests in 3 hops, 35% in 2 hops, less than 10% in 1 hop, and a few in more than 3 hops. FlowerCDN improves this distribution by honoring 70% of requests in 1 or 0 hops, and only 15% in 3 hops.

4.3 Impact of the View Size

We start dissecting the performance of Behave by analyzing hit rate and bandwidth consumption for different values of its view size. The results, depicted in Figure 3a show that Behave offers a significant gain over the local cache even with a very limited view size of 5 peers. The hit rate increases with larger view sizes of up to 25 peers (recall that the total number of peers is about 1000), but bandwidth consumption (shown in Figure 3b) also increases.

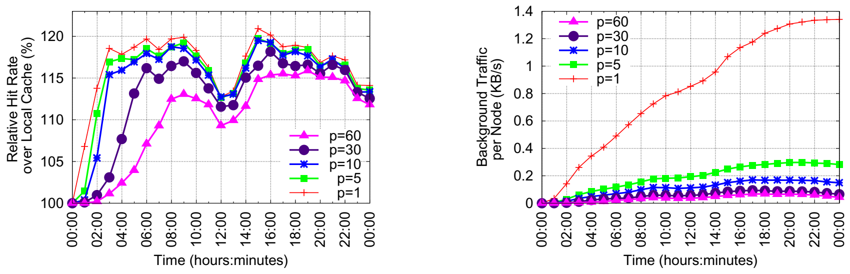
By analyzing the two plots, we can conclude that increasing the view size beyond 25 provides almost no benefit for a significant increase in bandwidth consumption. On the other hand, a view size of 25 appears to provide the best compromise between the two metrics.

Figure 3b also highlights Behave's effectiveness in adapting bandwidth consumption to the activity of users. Figure 1a shows that the number of visits per minute has a peak around 8am, then increases until 6pm, and finally decreases after 8pm: bandwidth consumption closely follows this pattern. Our profile-change threshold causes the frequency of gossip exchanges to follow the rate of changes in nodes' profiles, which is in turn proportional to the activity of users.



(a) Hit rate. (b) Bandwidth consumption.

Fig. 3. Impact of the view size in Behave



(a) Hit rate. (b) Bandwidth consumption.

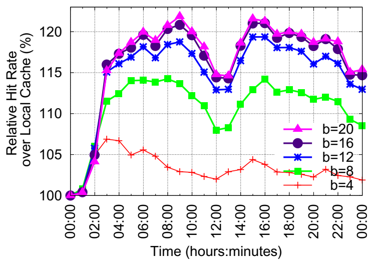
Fig. 4. Impact of the gossip period in Behave

4.4 Importance of the Gossip Period

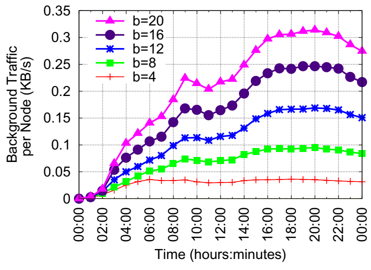
Next, we consider the impact of Behave's gossip period, p_g. Figure 4 confirms the expectations. Faster gossiping leads to better hit rates but also to higher bandwidth consumption. The impact affects not only the maximum hit rate but also the time Behave takes to converge, starting from an empty network. After convergence, gossip updates remain important because the behavior of users never reaches a steady state. New nodes start their activity and existing nodes visit new sites. Overall, the plots justify our default choice of a gossip period $p_g = 10$min. The corresponding bandwidth consumption remains below 0.2kB/s.

4.5 Bloom-Filter Behavior

Bloom filters play an important role in the performance of Behave. Figure 5 studies the impact of the number of bits per element, b, used by the interest-profile bloom filter. Results show that very small filters ($b = 4, 8$) provide unsatisfactory performance. However, the hit ratio does not change significantly when b grows beyond 12, while bandwidth consumption continues to grow. This justifies our choice of a default value of $b = 12$ for the interest-profile bloom filter corresponding to a false-positive probability of about $P = 0.003$. For the cache bloom-filter,

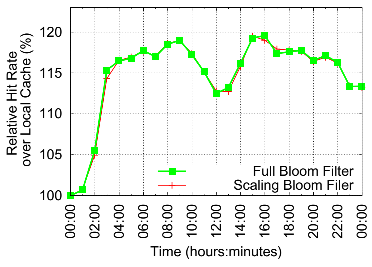


(a) Hit rate.

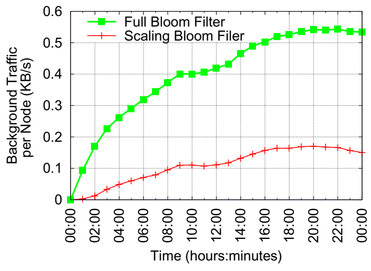


(b) Network consumption.

Fig. 5. Impact of bloom filter size in Behave



(a) Hit rate.



(b) Network consumption.

Fig. 6. Impact of scaling bloom filters in Behave

we stick to a value of $b = 20$ (false-positive rate $P = 6.7 \cdot 10^{-5}$) because false positives in the cache have a direct impact on the latency perceived by users: they cause an increase in cache misses as a result of failed requests.

Figure 6 highlights the advantage of periodically recomputing and resizing bloom filters. This prevents Behave from sending large bloom filters when profiles or caches contain only a few elements. The plot shows that scaling the size of bloom filters significantly reduces bandwidth consumption without affecting the hit rate. The increase in false positives with small bloom filters is in fact very limited. For example, with 12 bits per element, $p(\textsf{false positive}) = 3.14 \cdot 10^{-3}$ with 5000, while $p(\textsf{false positive}) = 3.29 \cdot 10^{-3}$ with only 5.

5 Related Work

Recent research has proposed a variety of solutions for peer-to-peer-oriented content-delivery networks. Squirrel [6] and Backslash [3] propose decentralized web caches that index content with a DHT. Their authors propose two strategies: replicate web objects within the DHT, or use DHT nodes as pointers to the stored data. We compared our approach with the latter strategy in Section 4.1.

CoralCDN [27, 28] uses instead peer-to-peer technology to organize a collection of servers without offloading tasks to user machines. Its authors acknowledge the limitations of standard DHTs and propose a modified DHT to implement their CDN. In a later paper [18], some of the authors propose a new completely decentralized-architecture that exploits signatures to guarantee content integrity. We adopt a similar signature-based approach in Behave.

FlowerCDN [5] combines a DHT with gossip-based communities of nodes that focus on specific content. Unlike our approach, they form these communities on a per-site and per-location basis. This means that the communities associated with two websites remain uncorrelated even if they may contain common nodes. This prevents FlowerCDN from exploiting semantic links between websites.

Maygh [7] takes advantage of the recent WebRTC and RTMP technologies to build a decentralized CDN without installing new software on clients. However, its approach requires intervention from website owners who must modify their websites in order to employ Maygh's technology. Moreover Maygh's architecture relies on coordinator nodes that must be deployed by each participating website.

SocialCDN [29] builds a collaborative cache that exploits explicit social acquaintances to aggregate peers for content distribution. This makes its approach limited to the case of distributed Online Social Networks. Behave, on the other hand offers a solution for a more general use case, even though it would be interesting to evaluate its performance in the case of online social networks.

Overall, only a few authors have focused on reducing lookup latency in the context of peer-to-peer CDNs or caches. The major example, Beehive [14], uses replication to achieve $O(1)$ lookup times in a DHT. Yet, its proactive approach focuses only on the most popular content. This makes Beehive unsuitable for optimizing the delivery of niche content. Behave, on the other hand, can perform particularly well on niche content thanks to the personalized approach provided by the Gossple similarity metric [13].

6 Conclusions

We proposed Behave, a peer-to-peer solution for building a content-delivery network based on the principle of Behavioral Locality. By adapting the concept of collaborative-filtering to decentralized caches, Behave provides zero lookup latency on about 50% of the content available through a DHT. This allows Behave to operate as a stand-alone solution, coupled with existing local caches, or in combination with a DHT.

The promising results we obtained by simulation encourage us to finalize the development of our Java and WebRTC prototypes and to explore novel research directions. First, we plan to integrate Behave with privacy and trust-based mechanisms. This would allow us to expand the applicability of Behave. Second, we envision combining our CDN architecture with non-web-oriented recommendations. This could provide benefits in both directions: more effective content delivery, and more precise recommendations. Finally we plan to study how our Behavioral cache can be integrated into existing CDN solutions.

Acknowledgments. This work was partially funded by the ERC SG GOSSPLE project (204742) (http://gossple.fr), the EIT/ICT-Labs AllYours project (CLD 13 021), and by the French National Research Agency (ANR) project SocioPlug (ANR-13-INFR-0003) (http://socioplug.univ-nantes.fr).

We would like to thank the Ircache project (National Science Foundation grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research for providing traces used in our evaluation. Finally, we are grateful to Mohammed Alaggan, Stephane Bortzmeyer, Arnaud Jegou, Julien Stainer and the Firefox "Necko" networking team, for useful comments and discussions.

References

1. Raspberry Pi, `http://www.raspberrypi.org`
2. FreedomBox, `https://freedomboxfoundation.org`
3. Stading, T., Maniatis, P., Baker, M.: Peer-to-peer caching schemes to address flash crowds. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 203–213. Springer, Heidelberg (2002)
4. Akamai, `http://www.akamai.com`
5. El Dick, M., Pacitti, E., Kemme, B.: Flower-cdn: a hybrid p2p overlay for efficient query processing in cdn. In: 12th International Conference on Extending Database Technology: Advances in Database Technology, pp. 427–438. ACM (2009)
6. Iyer, S., Rowstron, A., Druschel, P.: Squirrel: A decentralized peer-to-peer web cache. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 213–222. ACM (2002)
7. Zhang, L., Zhou, F., Mislove, A., Sundaram, R.: Maygh: Building a cdn from client web browsers. Image 70(40.3), 85–87 (2013)
8. Michiardi, P., Carra, D., Albanese, F., Bestavros, A.: Peer-assisted Content Distribution on a Budget. Computer Networks (February 2012)
9. Falkner, J., Piatek, M., John, J., Krishnamurthy, A., Anderson, T.: Profiling a million user dht. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, pp. 129–134. ACM (2007)
10. Crosby, S.A., Wallach, D.S.: An analysis of bittorrent's two kademlia-based dhts. Technical Report TR07-04, Rice University (2007)
11. Jimenez, R., Osmani, F., Knutsson, B.: Sub-second lookups on a large-scale kademlia-based overlay. In: 2011 IEEE International Conference on Peer-to-Peer Computing (P2P), pp. 82–91. IEEE (2011)
12. Ekstrand, M., Riedl, J., Konstan, J.: Collaborative Filtering Recommender Systems. Now Publishers (2011)
13. Bertier, M., Frey, D., Guerraoui, R., Kermarrec, A.-M., Leroy, V.: The gossple anonymous social network. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 191–211. Springer, Heidelberg (2010)
14. Ramasubramanian, V., Sirer, E.G.: Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In: Networked Systems Design and Implementation (NSDI 2004), San Francisco, USA, pp. 99–112 (2004)
15. HTTP Archive, `http://httparchive.org`
16. Kermarrec, A.M., Leroy, V., Moin, A., Thraves, C.: Application of Random Walks to Decentralized Recommender Systems. In: 14th International Conference on Principles of Distributed Systems, Tozeur, Tunisie (2010)

17. Boutet, A., Frey, D., Guerraoui, R., Jégou, A., Kermarrec, A.M.: WhatsUp Decentralized Instant News Recommender. In: IPDPS 2013, Boston, USA (May 2013)
18. Terrace, J., Laidlaw, H., Liu, H., Stern, S., Freedman, M.: Bringing p2p to the web: Security and privacy in the firecoral network. In: Proceedings of the 8th International Conference on Peer-to-Peer Systems, p. 7. USENIX Association (2009)
19. Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.L.: Networking named content. In: 5th International Conference on Emerging Networking Experiments and Technologies, pp. 1–12. ACM (2009)
20. Boutet, A., Frey, D., Jégou, A., Kermarrec, A.-M., Ribeiro, H.B.: FreeRec: an Anonymous and Distributed Personalization Architecture. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 58–73. Springer, Heidelberg (2013)
21. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (June 1999) Updated by RFCs 2817, 5785, 6266, 6585
22. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communication of the ACM 13(7), 422–426 (1970)
23. Almeida, P.S., Baquero, C., Preguiça, N., Hutchison, D.: Scalable bloom filters. Information Processing Letters 101(6), 255–261 (2007)
24. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: Building a better bloom filter. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 456–467. Springer, Heidelberg (2006)
25. Rosenberg, J.: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard) (April 2010) Updated by RFC 6336
26. Boucadair, M., Penno, R., Wing, D.: Universal Plug and Play (UPnP) Internet Gateway Device - Port Control Protocol Interworking Function (IGD-PCP IWF). RFC 6970 (Proposed Standard) (July 2013)
27. Freedman, M.: Experiences with coralcdn: A five-year operational view. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, p. 7. USENIX Association (2010)
28. Freedman, M., Freudenthal, E., Mazieres, D.: Democratizing content publication with coral. In: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation, p. 18. USENIX Association (2004)
29. Han, L., Punceva, M., Nath, B., Muthukrishnan, S., Iftode, L.: Socialcdn: Caching techniques for distributed social networks. In: 2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P), pp. 191–202 (2012)