Available online at www.sciencedirect.com
SCIENCE DIRECT
J. Parallel Distrib. Comput. 65 (2005) 1483 – 1496
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Adaptive schemes for distributed web caching

Spiridon Bakiras[a], Thanasis Loukopoulos[a], Dimitris Papadias[a,*], Ishfaq Ahmad[b]

[a]Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
[b]Department of Computer Science & Engineering, University of Texas at Arlington, TX, USA

Received 8 September 2003; received in revised form 3 February 2005; accepted 23 May 2005
Available online 26 July 2005

Abstract

In distributed web caching architectures, institutional proxies take advantage of their neighbors' contents in order to reduce the number of requests forwarded to the server. Intuitively, the maximum benefit from this cooperation is expected when the proxies that exhibit similar requests are grouped together. The current practice is to follow a static and manual configuration of neighbors. Such an approach has a number of drawbacks: (i) static allocation may not determine the best neighbors, especially if global knowledge of the participating proxies is not available, (ii) a manual allocation places significant administrative burden, (iii) static schemes are insensitive to changes in access patterns, and (iv) they cannot deal with the introduction of new, potentially useful, proxies. In this paper, we propose a set of algorithms that allow proxies to independently explore the network for better neighbors and continuously update their configuration in an adaptive fashion. The simulation experiments illustrate that dynamic neighbor reconfiguration leads to significantly higher hit ratios compared to the static approach. Although some researchers in the past have recognized the need for adaptive caching, to the best of our knowledge this is the first study to propose concrete algorithms and evaluate their efficacy.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Distributed caching; Web proxies; Cache digests; Squid

1. Introduction

Proxy caching has emerged as a primary technique to reduce the latency experienced by end-users when downloading web pages. Its apparent success is based on the premise that sharing cached contents in a multi-user environment leads to increased hit ratio and, consequently, better performance. The same premise led to the development of hierarchical [CDN+96] and distributed caching [TDV+99].

A caching hierarchy is defined through parent–child and sibling relations among the participating proxies. In the basic scheme introduced by the *Harvest* system [CDN+96], client requests arrive at the lowest level and misses are forwarded to the upper levels until the root node is reached. If the root is unable to satisfy a request, the web server is contacted. Although hierarchies usually result in high hit ratios for the intermediate and topmost nodes, they possess two main drawbacks: (i) the benefit (in terms of response time) for end-users is not always possible (especially if the topmost cache lies behind a slow link), (ii) upper level nodes may become overloaded. For these reasons, the number of levels is commonly restricted to three, i.e., institutional, regional and national.

Distributed caching can be viewed as a step forward in an attempt to overcome the deficiencies of hierarchies. In a purely distributed scheme, institutional proxies cooperatively satisfy user requests without the presence of regional and national caches being necessary. Hybrids between distributed and hierarchical caching have also been proposed (e.g., in [TDV+99] where the hierarchy is only used for propagating metadata information concerning content locations). *Squid* [WWW1], which is the successor of Harvest, provides enough versatility in the cache configuration to

* Corresponding author. Fax: +852 2358 1477.
E-mail addresses: sbakiras@cs.ust.hk (S. Bakiras), luke@cs.ust.hk (T. Loukopoulos), dimitris@cs.ust.hk (D. Papadias), iahmad@cse.uta.edu (I. Ahmad).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.05.020

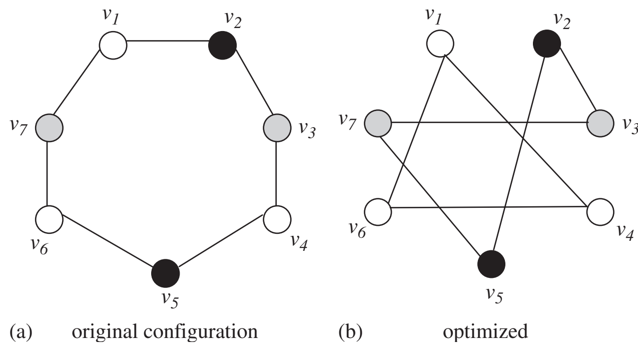

Fig. 1. A scenario of original and optimized configurations: (a) original configuration; (b) optimized.

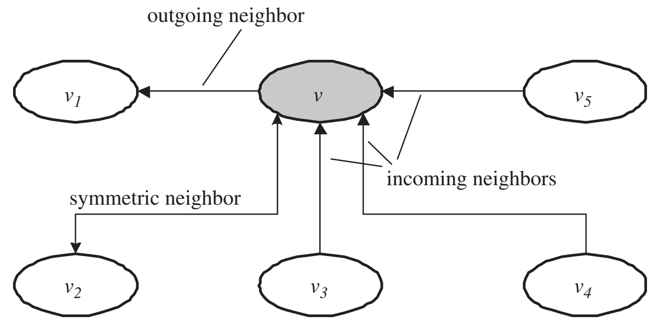

Fig. 2. Unidirectional and bidirectional neighbor relations.

account for hybrid architectures, and includes a dedicated protocol for inter-proxy querying (*Internet Cache Protocol* ICP [WC97]). In the basic scheme, when a miss occurs, the cache broadcasts the query to its neighbors and retrieves the page from the first one that replies positively. If none of the neighbors has a cached copy, the request is forwarded either to the parent cache or to the web server. An alternative is provided by *cache digests* [RW98], where proxies exchange periodically their directory information in the form of compressed hash arrays. In this way, a proxy checks the digests of its neighbors (stored locally) and forwards the request only to the neighbors that have cached the page.

Ideally, a distributed caching scheme should achieve the hit ratio of a single proxy acting over the combined population of all the proxies. In practice, significant performance degradation occurs since users from different institutional proxies can exhibit arbitrarily diverse surfing behaviors. Furthermore, the number of neighbors for a single proxy must be restricted, due to local resource limitations and in order to avoid overloading the network with redundant messages. Therefore, enabling proxies to select the best neighbor candidates in an automatic and dynamic way becomes of paramount importance, and is the main motivation for our work.

If possible, a proxy should only make neighbors other nearby (in terms of network latency) proxies with similar access patterns. As an example, assume the simple network of Fig. 1(a) where the nodes correspond to proxies and the edges to neighborhood relations (each proxy has two neighbors). The similarity of node colors corresponds to similarity of cache contents (the largest difference exists between black and white proxies). Obviously, this configuration is not very beneficial since the neighbor proxies are usually dissimilar. On the other hand, the clustering of Fig. 1(b) is more useful for content sharing, because similar proxies are grouped together.

An optimal grouping of proxies into neighborhoods is difficult to determine for several reasons: (i) global information about the cache contents is not available, (ii) access patterns change and, as a result, the neighborhood graphs need to be updated continuously, and (iii) each proxy should independently take decisions about its neighbors in order to maximize its hit ratio. In this paper, we propose a set of distributed algorithms that dynamically group proxies into neighborhoods. The algorithms estimate the potential for content sharing, based solely on the information available at each proxy. Since it is not feasible to have knowledge about all the participating proxies, an *exploration* step discovers caches with similar access patterns.

The problem can be thought of as *second level caching*, where the cached objects are the best neighbors of each proxy v. When v determines that a proxy v_i (not currently in v's neighborhood) could provide a large number of hits, it adds v_i to the list of its best neighbors by evicting the least beneficial existing neighbor. We distinguish two cases: in the first case, called *unidirectional* (or asymmetric) caching, we assume that although a proxy has a maximum number k of *outgoing* neighbors (to which it forwards its requests), it serves an arbitrary number of *incoming* proxies. As an example consider Fig. 2, where v has only $k = 2$ outgoing neighbors (v_1 and v_2), but receives requests from four incoming proxies (v_2 to v_5). In the second case, called *bidirectional caching*, all neighborhood relations must be symmetric (e.g., v and v_2 in Fig. 2), i.e., a proxy has a total of k neighbors, each of which is both incoming and outgoing.

The rest of the paper is organized as follows. Section 2 presents the related work on distributed caching. Sections 3 and 4 describe the algorithms for unidirectional and bidirectional caching, respectively. Section 5 presents the experimental results, and Section 6 concludes the paper with a discussion about the future directions.

2. Related work

Several papers have focused on quantifying the potential gains of distributed caching. In [KS98] the authors analyze traces from Bell Labs reporting that the performance improvement for an ideal cooperative scheme is significant, but varies considerably depending on the day of the traces. They also observe that the cooperation of only a small set of proxies has a high impact on performance. The study

in [WVS+99] analyzes traces from a large number (tens of thousands) of end-clients, and provides an analytical model to predict the system's behavior. The authors conclude that the largest benefit from distributed caching is expected when the number of clients is relatively small.

In [DR01], the authors estimate the average response time in hierarchical and distributed caching architectures, and conclude that the speedup from distributed caching is higher than that of hierarchical caching. In [RSB99] the authors break the response time into connection and transmission time. They suggest that distributed caching accounts for larger connection times, but smaller transmission delays, since lower level links are usually not congested. They also argue that a hybrid architecture comprising multiple caching meshes organized hierarchically, achieves the best performance. The work in [BCZ98] extends the distributed caching scheme in the active network context, by proposing the placement of small-sized caches in network switches. The authors provide trace-driven simulation results and an analytical model to evaluate their proposal.

Another important issue of distributed caching is how to locate a specific page. The solution in [TDV+99] proposes the use of hints that are simple records of the form ⟨*object_id*, *closest_neighbor*⟩, cacheable at each proxy. A static hierarchy is used for hint propagation. Another approach in [RCG98] proposes a centralized control that keeps information about the contents of all proxies. *Cachemesh* [W97] employs URL routing tables, maintained in a way similar to the IP routing tables, for redirecting requests to appropriate caches. The *Cache Array Routing Protocol* (CARP) [VR98] splits the URL space using hash functions and allocates different portions to each proxy, taking into account their processing capacity.

In general, existing work seems to agree on the benefits of sharing caches, although the level of the gain depends on several parameters (e.g., proxy configuration, access patterns, cache sizes, network latency, etc.). Currently, the most popular system for distributed caching is *Squid* [WWW1] according to which proxies are manually configured in fixed neighborhoods. The current version of Squid implements *cache digests* [RW98,FCA+98], which are compact representations of cache contents based on Bloom filters [B70]. Fig. 3 illustrates a simple example, when the cache digest is an array of m bits which are initially set to 0. When a new page W is cached, its MD5 signature [R92] is hashed using n hashing functions $h_1, \ldots, h_n$ (each with range $\{1,\ldots,m\}$), and the bits at positions $h_1(W), \ldots, h_n(W)$ are set to 1. In addition, there exists a second array of counters. The insertion of W will increase the counters at positions $h_1(W), \ldots, h_n(W)$. When W is evicted from the cache, the value of each counter in these positions is decreased by one. If some counter becomes 0, the corresponding bit in the cache digest is also set to 0.

The counter array is only kept at the corresponding proxy, whereas the cache digest (bit array) is sent to all the neighbors. When a local miss occurs at proxy v, v redirects the request to a neighbor proxy v_i that has the required page (according to the digest of v_i stored locally at v), thus avoiding the extra latency introduced by ICP. In order for v to locate a page W in v_i's digest, it only needs to check the bits at positions $h_1(W), \ldots, h_n(W)$: (i) If all bits are 1, v conjectures that W is in v_i's cache, although there is a probability of a *false positive*. False positives occur when multiple pages set the same bits. The trade-off between space overhead and percentage of false positives is tuned by choosing appropriate values for parameters n and m. (ii) If any of the bits at positions $h_1(W), \ldots, h_n(W)$ is 0, v concludes that W is not in v_i's cache.

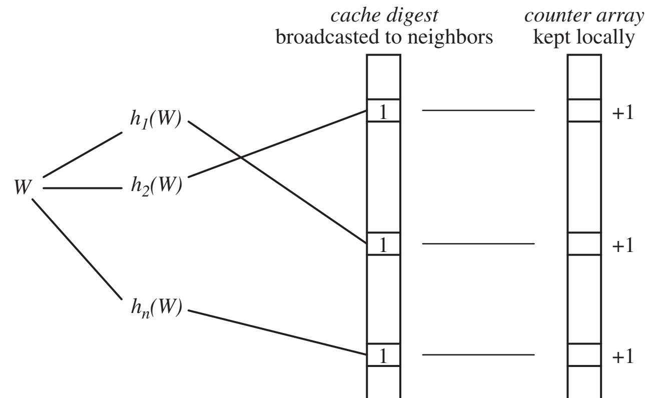

Fig. 3. Example of cache digest.

Since v_i's digest is not necessarily up-to-date, a *false miss* may occur if v_i has cached W after it sent its digest to v. Similarly, an outdated digest may also cause *false hits*, if a page that appears in the remote digest has meanwhile been evicted from the local cache. As shown in the simulation results of [FCA+98], cache digests achieve significant bandwidth savings compared to ICP querying. Furthermore, the authors observe that even with infrequent summary updates the performance loss (due to false misses or false hits) is marginal. In particular, [FCA+98] concludes that updated digests should be propagated to neighbors only after 1–10% of the cache contents change. Dykes and Robbins [DR01] suggest that updates should happen on a daily basis when the network traffic is low (e.g., during the night). Due to the apparent benefits of summaries, we include them in our design.

In terms of concept, the work reported in [ZMN+97] is perhaps the closest work to ours. The authors propose a hybrid architecture, with proxies participating in a hierarchy of overlapping multicast groups. Such a scheme introduces the problem of determining when a proxy should enter or leave a group. The paper motivates the need for a dynamic method, but does not provide any concrete algorithms or performance evaluation. Furthermore, the neighbor selection problem as defined in our paper differs in certain ways, i.e., we (i) consider a fully distributed architecture, (ii) employ cache digests, (iii) distinguish between unidirectional and bidirectional caching, and (iv) do not assume multicast ca-

pabilities. The scope of our approach is also different. While [ZMN+97] proposed a novel (at that time) architecture, we aim at exploring whether there is room for performance improvement in current distributed caching systems by moving from a static neighborhood grouping to a dynamic one, and illustrate methods to exploit the potential.

3. Unidirectional caching

In this section we describe the proposed algorithms for neighbor selection under the unidirectional model (i.e., asymmetric neighborhood relations). The goal is to allow each node to dynamically update its neighborhood list in order to maximize the number of hits from other proxies. Neighboring nodes exchange their cache digests so that when a miss occurs, the missing page may be obtained directly. Since the most useful neighbors vary continuously with the access patterns, the system should include a mechanism to replace the least beneficial neighbors with new ones that may provide more hits.

Subsequently we propose two techniques: the first one is an adaptation of the *Least Recently Used* (LRU) strategy, while the second one extends the *Least Frequently Used* (LFU) paradigm by taking into account the special characteristics of the problem.

```
Algorithm Serve_Request (Page: W)
1. if W is cached locally then serve request and return
2. l=list of neighbors whose digests contain W sorted by network latency
3. while l is not empty
4.    remove the closest neighbor v_i from l
5.    query (get_page, W, v, v_i)
6.    If hit then send W to client, update recency of v_i and return
7. end //while
8. forward the request to the web server // all neighbors produced a miss
9. for each neighbor v_i query(explore, W, v, v_i) with probability a
10.l_2=collect-exploration-result(W,time-out) // l_2 is a list of proxies that have cached W
11.let v_new be the closest node in l_2
12.if latency(v_new)<latency(server) then
13.   get cache digest of v_new
14.   v_new becomes the most recent neighbor (possibly by evicting the least recent one)
end Serve_Request

Algorithm Process_Query (String: op_code, Page: W, Node: v (originator), Node: v_i (current node))
1. CASE (op_code)
2. get_page:
3.    if W in cache then send W to v and signal hit = true
4.    else (W not in cache) signal hit = false
5. explore:
6. if W in cache then notify v and return
7. if limit of hops has been reached then return
8. for each neighbor v_j of v_i
9.    if v_j's digest contains W then query(explore, W, v, v_j)
10. else // v_j's digest does not contain W
11.         query (explore, W, v, v_j) with probability a
end Process_Query
```

Fig. 4. Pseudo-code for LRU.

3.1. Unidirectional LRU

Assume that a node v has k neighbors $v_1, \ldots, v_k$. When a miss occurs, v sends the request to one of $v_1, \ldots, v_k$ that has cached the page (according to the cache digests). If multiple neighbors can serve the request, the one (denoted by v_i) that is closer (in terms of latency) is chosen. When v_i returns the page it becomes the most recent neighbor. In case of a false hit/positive the process is repeated for all neighbors whose digest contains the page.

If none of the neighbors can provide the page, v sends the request to the server and at the same time initiates an *exploration* process. The goal of the process is to identify other nearby proxies that have the page, since such proxies may be beneficial for subsequent requests. The pseudo-code for the algorithm is illustrated in Fig. 4. The pseudo-code distinguishes two cases: *serve_request*, which corresponds to the situation that a page W is requested by a client, and *process_query* where a query is received from another proxy. This query can be (i) a request for a page, or (ii) an exploration query.

The exploration process deserves further elaboration. When node v receives a client request for a page W not cached locally or in its neighbors, it sends an exploration query to all the neighbors v_i with probability $a \leqslant 1$. A node v_i that receives an exploration query first checks its own cache and, if it contains W, it replies directly to v. Otherwise, v_i forwards the query to all neighbors whose digest contains W, and to the remaining ones with probability a.

The forwarding process continues until a maximum number of hops h is reached. Like parameter a, the value of h adjusts the trade-off between the extent of exploration and traffic overhead. The original node v accumulates responses from proxies caching W, until a time-out period is exceeded. Then it makes as its most recent neighbor, the proxy v_{new} that contains W and has the lowest network latency. If the list of neighbors is full, the least recent neighbor is evicted. Note that due to false misses, v_{new} may already be a neighbor of v, in which case v_{new}'s digest is updated (no neighbor is evicted). If no response arrives before the time-out interval the recency status remains unchanged.

There is no forwarding of actual page requests since they are satisfied either at: (i) the proxy v where they arrived, (ii) a first degree neighbor of v, (iii) the web server. If v determines that no neighbor proxy contains W, it sends directly the request to the appropriate server without waiting for the results of the exploration process. This is done in order to minimize the delay experienced by end-users. Further delay occurs only in the case of consecutive false hits/positives, since our policy forwards the request sequentially to all the neighbors before sending it to the server. Nevertheless, with a reasonable size of cache digests this situation is highly unlikely (see Section 2). Furthermore, the latency for satisfying the request from the server is expected to be much higher than that of issuing queries to nearby neighbors.

A final remark concerns some implementation issues. For querying, the *http get* method provides the functionality of the *get_page* op_code. We can implement *explore* in ICP, by adding a new op_code in one of the unused slots. The payload of the new op_code should include, apart from the URL, the number of hops that the query has traveled so far and the id of the proxy where it originated, so that the receiver can answer directly to it. The algorithm also requires the proximity (latency) computation between proxies, in order to choose the fastest neighbor (in case of multiple available choices). For existing neighbors, the proximity can be calculated by taking a weighted average of the past experienced latency [GEC+99]. For proxies accessed (by exploration) for the first time, the latency can be estimated by measuring the RTT using the *ping* utility. In our simulations we assume that knowledge of the closest neighbor is always available.

3.2. Unidirectional LFU

A potential problem with LRU is that it imposes network overhead due to the frequent reconfiguration of the neighbors and exchanges of digests. In addition, LRU may quickly replace some "good" neighbors that do not provide hits for a short period of time, although they are beneficial in the long run. The second strategy, unidirectional LFU, aims at overcoming these problems by collecting statistics and performing reconfiguration if certain conditions hold.

The pseudo-code for LFU is very similar to that of Fig. 4 and thus is not included here. The algorithm keeps a few hit counters at each proxy, which maintain the *positive responses* (page retrievals or exploration hits) from other proxies. Each page retrieval or exploration hit received by node v increases at most one hit counter (of the fastest node) in v, even though multiple proxies may respond. Thus, closer neighbors are favored and the network latency is implicitly included in the number of hits. Moreover, since a proxy needs to obtain only one copy of a page (ideally the closest available) the existence of other copies yields no local benefit. This "implicit-latency" approach is followed by all algorithms.

When reconfiguration is performed, the new neighborhood of a proxy v is defined as the set of k nodes that provided the largest number of positive responses to v. Some of these nodes may be already in the neighborhood of v, and no special action is required. For the rest of the nodes, v requests and maintains locally their cache digests by evicting the digests of aborted neighbors. The exchange of digests implies that digests are usually up to date. In some situations, however, it is possible that a proxy v_i will remain a neighbor of v for a long period of time. Special care must be taken in order to update its digest because the methods available for Squid are now inapplicable. Recall that in Squid all the neighbors of v_i obtain (simultaneously) the same version of v_i's digest. Thus, v_i can decide locally when the digest is outdated and broadcast the new version to all its neighbors. On the other hand, in our methods this decision is taken asynchronously at each receiving node (since nodes have different versions of the digest depending on when they configured v_i as a neighbor); a proxy will ask for a new digest from a neighbor, if the percentage of false misses from this neighbor (discovered through exploration) exceeds a threshold.

An interesting issue regards the appropriate conditions for reconfiguration. At one extreme, if reconfiguration occurs after every positive response, the strategy will transform to LRU. At the other extreme, if reconfiguration is very infrequent, LFU will behave like a static scheme. In order for the statistics to be meaningful, reconfiguration is initiated after a number l of positive responses has been collected. Local hits, or queries that do not yield any exploration results, do not provide any information about the contents of other proxies; only page hits from neighbors or exploration hits are useful for the computation of neighbors. When l (good values of l are determined experimentally) is exceeded, a non-neighbor v_j, will replace a neighbor proxy v_i, if the value of the counter for v_j is above r% of the corresponding value for v_i. If, for instance, $r = 100$, v_j will replace v_i, if it provides more positive answers. In practice, since neighbors are favored because they are requested first, the value of r should be lower.

Notice that since we aim at maximizing the hit ratio from other proxies, we only take into account the number of hits and not the page sizes. Intentionally, we tried to keep the neighbor replacement policy as simple as possible by adopting the well-known LRU and LFU paradigms, because our goal is to demonstrate the advantages of adaptive caching in general, and not of the individual policies. We also experimented with alternative caching strategies (e.g., based on the *Greedy-Dual* algorithm [CI97]) that consider additional parameters such as detailed benefit and latency measures, but found that the additional gains (if any) are negligible.

4. Bidirectional caching

In unidirectional caching, although a proxy may send its request to a set of k neighbors at any given time, it may serve requests from an arbitrary number of proxies. As an example consider one large cache v surrounded by numerous small ones. It is possible (and verified by our experiments) that most of the small proxies will attach to v, possibly overloading it with requests. The assumption that v will accept to serve all these proxies may be too strong in practice, especially if the proxies to be served belong to different organizations. In this section, we propose bidirectional extensions of LRU and LFU that permit only symmetric neighborhood relations.

Bidirectional LRU (B-LRU for short) is similar to its unidirectional counterpart. The difference is that when node v_i (that provides an exploration hit to node v) receives an in-

vitation to become a neighbor of v, it evicts one of its existing neighbors (let v_j) to make space for v. Proxies v and v_i exchange digests and each becomes the most recent neighbor of the other. The evicted proxy v_j should also evict v_i, since the neighborhood relation is now symmetric. In order to do this, v_j inserts v_i in a "replacement" list, until it finds a substitute. The digests of proxies in this list are only used for exploration, but not for actual page requests. The reason that v_i cannot evict v_j immediately is explained with the following example. Assume that v_j has a small number of neighbors, all of which decide to remove v_j from their neighborhood within a (short) time interval in which v_j did not receive any requests (and, therefore, did not perform exploration). If the neighbors were evicted right away, then v_j would have no digests (or neighbors) to initiate exploration and would remain isolated from the rest of the proxies.

B-LRU is more adaptive than unidirectional LRU in the sense that once v "discovers" v_i and configures it as a neighbor, v_i may also start immediately benefiting from the collaboration (without having to wait until it discovers v through its own exploration process). On the other hand, if v is not beneficial to v_i, it may decrease v_i's hit ratio by evicting some other, potentially more advantageous, neighbor of v_i. Furthermore, B-LRU is expected to increase the number of neighbor updates and the overhead of digest exchanges because of the "propagation" effect, e.g., the change of neighborhood in v, causes a change in v_i, which in turn causes a change in the neighborhood of v_j. In order to alleviate this overhead we also develop a bidirectional version of LFU.

According to B-LFU each proxy v collects statistics for a number l of positive responses. After this period it determines the set of k best proxies $v_i (i = 1, \ldots, k)$, which will become its new neighbors in the same way as unidirectional LFU. Some of these proxies may be already neighbors, in which case no special action is required. The rest will exchange digests with v. As in the case of B-LRU, proxies will have to evict some previous neighbors to accommodate new ones. The pseudo-code of Fig. 5 illustrates the actions performed by the original proxy (v), the new neighbors of v (e.g., v_i), and the evicted proxies.

In case of an invitation, the invited proxy v_i resets its statistics, meaning that it will attempt to perform reconfiguration only after it gathers l new positive responses (provided of course that v_i does not receive an invitation before). In case of an eviction, the evicted proxy v_j sets the number of hits from the evicting node (v) to zero, but does not restart the counter of positive responses (i.e., reconfiguration of v_j will not be delayed). The statistics of v are reset in order to avoid the situation in which v_j reconfigures v as a neighbor, which could possibly incur a new eviction for v_j. To prevent the propagation effect, v_j does not immediately substitute v with a new neighbor, but waits until the end of reconfiguration period.

Notice that bidirectional caching is a constrained version of the corresponding unidirectional problem. As a result, bidirectional methods are not competitors of unidirectional techniques, but alternatives applicable to symmetric neighborhood relations. We expect that real life situations are somewhere in the middle, i.e., a large proxy would serve requests from an arbitrary number of other proxies in the same organization, but only a limited number of external proxies. Such a configuration would require a *hybrid* of unidirectional (for proxies in the organization) and bidirectional (for external proxies) caching, which can be easily implemented using the proposed techniques.

```
Algorithm Reconfigure (Node: v)
1. l_old = list of (k) proxies in the old neighborhood
2. l_new = list of (k) most beneficial proxies
3. for each proxy v_old in l_old but not in l_new
4.         remove v_old's digest
5.         send eviction(v, v_old)
6. for each proxy v_i in l_new but not in l_old
7.         send invitation(v, v_i)
8.         send digest(v, v_i)
end Reconfigure

Algorithm Process_Invitation (Node: v (originator), Node: v_i (current-invited-node))
1. remove digest of the least beneficial neighbor v_j according to current statistics
2. send eviction(v_i, v_j)
3. store digest of v
4. send digest(v_i, v)
5. reset statistics
end Process_ Invitation

Algorithm Process_Eviction (Node: v (originator), Node: v_j (current-evicted-node))
1. insert v's digest in replacement list
2. reset v's statistics
end Process_Eviction
```

Fig. 5. Proxy actions for B-LFU.

5. Experiments

We evaluate the proposed algorithms as follows. Section 5.1 describes the traces used in all experiments. Section 5.2 compares unidirectional LRU with (unidirectional) Squid variants, in order to confirm the viability of adaptive caching. Section 5.3 measures the performance of LFU against LRU. Section 5.4 evaluates the alternative bidirectional schemes, and Section 5.5 provides some insight on the behavior of different strategies. Finally, Section 5.6 summarizes the results.

Although the traces used originated from real proxies, we did not have any information about the network topology. Therefore, we assume a fully connected network where the inter-proxy (one-way) latency follows a Gaussian distribution with mean 70 ms and standard deviation 20 ms. Values below 10 ms and greater than 130 ms were cut off. These numbers correspond to proxies that are within a small geographical area (e.g., one city). For instance, one may con-

Table 1
Statistics of NLANR traces

	startap	*bo*2	*bo*1	pa	sv	sd	uc	pb	rtp
Tot. Size (GB)	1.63	2.16	3.03	3.06	6.93	6.97	7.76	20.38	33.53
Reqs (Millions)	0.46	0.35	0.42	0.76	2.35	1.29	0.79	3.09	6.29
Dist. Pages (Mil.)	0.22	0.24	0.28	0.35	0.80	0.73	0.51	1.58	2.98
Avg. Pg. Size (KB)	7	9	10	9	9	10	15	13	11

sider that two proxies with a small inter-proxy latency (i.e., around 10–50 ms) belong on the same LAN, while others reside on distant LANs. The (one-way) latency between proxies and web servers is fixed to 1 s in order to simulate the situation where fetching pages from proxies is much faster than doing so from the servers. This assumption is valid, since (i) requests for “local” servers do not yield any ICP queries, and (ii) the ICP_OP_SECHO opcode may be used to identify whether the server is closer than the neighbors. As a measure of performance we employ the number of neighbor hits, i.e., local misses served by the (1st degree) neighbors, because it is less sensitive than other measures (e.g., average response time) to the (artificial) network latencies.

For the implementation of the Squid simulations we followed the guidelines of [RW98]. Each proxy broadcasts a new digest version to its neighbors whenever the cached contents change by 1%. On the other hand, our methods exchange digests when the percentage of (identified) false misses exceeds 1%. In all simulations, the cache for each proxy is equal to 10% of the total size of the locally requested objects (this is common practice in related work, e.g., [FCA+98]). The (local) page replacement strategy for all proxies is LRU.

5.1. Datasets

Real data: We collected traces from the 10 available proxies of the *National Laboratory for Applied Network Research* (NLANR [WWW2]). These proxies are based on the Squid software and are located throughout the United States. Their aim is to provide hierarchical caching services to organizations and individuals. The traces depict all requests between 15/11/01 and 18/11/01. We decided to exclude the *sj* proxy from our experiments, since it accounts for very light and dissimilar workload compared to the rest. Moreover, only HTTP requests with the GET method are considered, since only this type of requests may trigger an ICP query. URLs containing “cgi-bin”, “.asp” and “?” substrings are excluded as un-cacheable objects. The same is true for requests with a result code TCP_CLIENT_REFRESH_ MISS, since they account for a *no-cache pragma*, control command. Finally, we deleted requests for partial content (status 206) and requests that resulted in 0 byte transfers. This methodology has been suggested in previous related work [DRJ01]. The statistics for the remaining pages are summarized in Table 1.

Although traces from institutional proxies could be more appropriate in our study, we were unable to collect a sufficient number of them. Nevertheless, we believe that recreating the behavior of the topmost proxies in the NLANR hierarchy, is still sufficient for illustrating the main merits of the proposed strategies and providing useful insight. It is reasonable to expect similar or higher performance gains for institutional proxies where the sharing potential is higher.

Synthetic data: In order to test how the parameters of the algorithms and the network size affect performance, we created two synthetic datasets representing requests for 45 proxies. In the first set (SYNTH I), each of the 9 initial NLANR traces was split into 5 equal parts/proxies. Every request was sequentially assigned to one of the 5 proxies in a round-robin way (a similar method was followed in [W02]). Thus, the proportional size differences of the initial NLANR traces were also preserved in SYNTH I. The second dataset (SYNTH II) was created again using the round-robin method, but the larger proxies were split in more parts in order to minimize the size differences of the resulting 45 proxies. Experiments with SYNTH I aim at evaluating performance and scalability in an “expanded” NLANR hierarchy. SYNTH II approximates better the case of institutional level proxies where cache size is not expected to vary significantly. Whenever the results are similar, we only present SYNTH I. We were unable to follow the most intuitive approach of splitting the requests of the initial trace according to the origin IP address, since the anonymizer used by Squid (i.e., the process that modifies the IP address before updating the log file) does not produce consistent IP addresses across multiple days.

5.2. Unidirectional LRU vs. static methods

In this section, we compare unidirectional LRU with static alternatives. We start with NLANR (9 proxies) and continue with the synthetic datasets. The parameters of LRU are set as follows: probability to send an exploration query to a neighbor $a = 0.5$, maximum number of hops for exploration $h = 2$, number of (outgoing) neighbors $k = 3$. We measure performance in terms of neighbor hits against two unidirectional Squid configurations obtained as follows: (i) we executed 30 experiments using random static configurations where each proxy has 3 outgoing neighbors and an arbitrary number of incoming ones; (ii) for each execution we counted the total number of neighbor hits; (iii) the con-

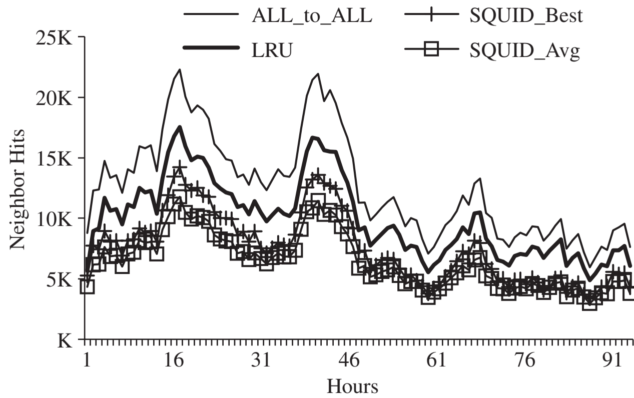

Fig. 6. The number of pages obtained from neighbors.

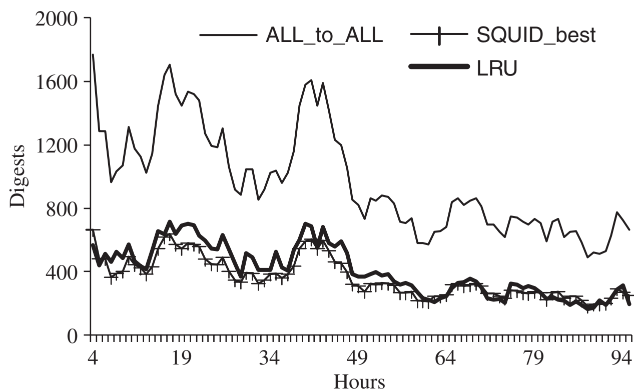

Fig. 7. The number of digests exchanged.

figuration that provided the mean of the total hits (i.e., the 15th best configuration) is *Squid_average*; (iv) the best (of 30) configuration is *Squid_best*. We also include the maximum number of neighbor hits that can be obtained if all proxies are connected (*All_to_all*). Fig. 6 shows the sum of hits of all proxies per hour (traces of 4 days—96 h).

LRU achieves a significant increase in the neighbor hit ratio compared to both static schemes with the same number of neighbors. This is expected since it dynamically modifies the initial configuration according to the access patterns. Its difference from the optimal hit ratio (*All_to_all*) is not large considering the limited numbers of neighbors (3) and exploration hops (2).

The second experiment (Fig. 7), illustrates the number of digests exchanged per hour. We only include *Squid_best*, because all static configurations result in more or less the same frequency of exchanges. Since in the first few hours there exist a lot of exchanges until the caches get full, we only show the results after the 4th hour.

The optimal (*All_to_all*) method is very expensive, since each proxy sends its updated digest to all the other eight proxies. Rather surprisingly, the overhead of LRU is similar to that of Squid. In LRU, a digest is sent from v_i to v_j when (i) v_i becomes a neighbor of v_j or (ii) v_j discovers a false

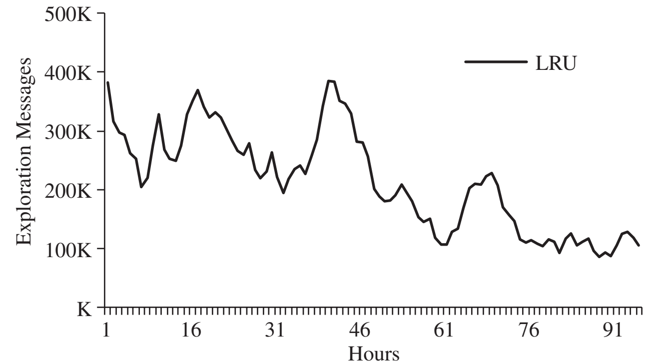

Fig. 8. The number of exploration messages.

miss in an existing neighbor v_i. In practice, the second case may be ignored since it is very infrequent. Therefore, essentially Fig. 7 implies that the number digests exchanged due to neighbor changes (in LRU) is more or less the same as the number of broadcasts in Squid (when the update threshold is 1%). We will explore this point further and study the effect of the network size in subsequent experiments with synthetic datasets.

In addition to digest exchanges, LRU (and all our methods) impose the overhead of exploration messages. Fig. 8 shows the total number of these messages per hour. Notice that the trend of the line is closely related to the number of neighbor hits (with peaks around the 16th and 40th hours), indicating increased exploration activity when the access patterns change substantially. Furthermore, given that the size of each message is several orders of magnitude smaller than that of a digest, the bandwidth overhead of digest transfers dominates that of exploration messages. In particular, the typical size of an exploration message is around 100–200 bytes (40 bytes for the TCP/IP headers, a few bytes for the protocol specific information, plus the size of the request string, i.e., 1 byte per character) and is independent of the proxy's cache size. On the other hand, the size of a digest may grow to large values, depending on the cache size of the proxy. For instance, if a proxy stores 100 thousand pages, the size of the digest would be 100 KB (assuming 8 bits per entry, which is used in Squid). For 1 million pages, however, the size of the digest becomes 1 MB, which is already four orders of magnitude larger than the exploration message (without measuring the TCP/IP overhead). With disk space becoming extremely cheap these days, it is safe to assume that the size of the digests will grow several orders of magnitude larger than a single exploration message.

Next, we use the dataset SYNTH I (45 proxies) to test the generality of the first observations. Notice that by splitting the contents of a proxy in smaller parts (i.e., the process that we followed to create the synthetic datasets) the total number of neighbor hits will increase since some local hits (i.e., at the same proxy) will now become neighbor hits. However, it is practically impossible to determine the actual number of neighbor hits since the size of the network is prohibitive for

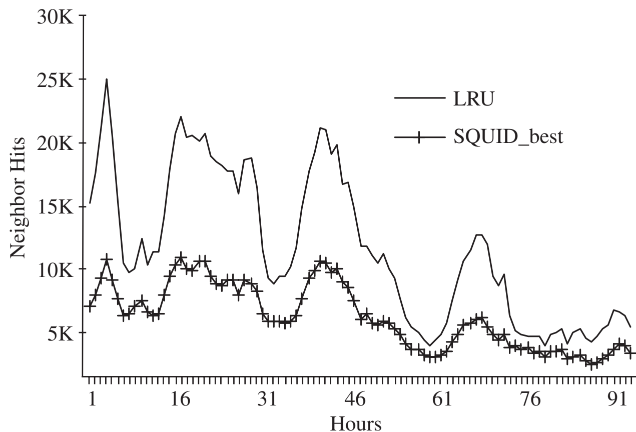

Fig. 9. The number of pages obtained from neighbors (SYNTH I).

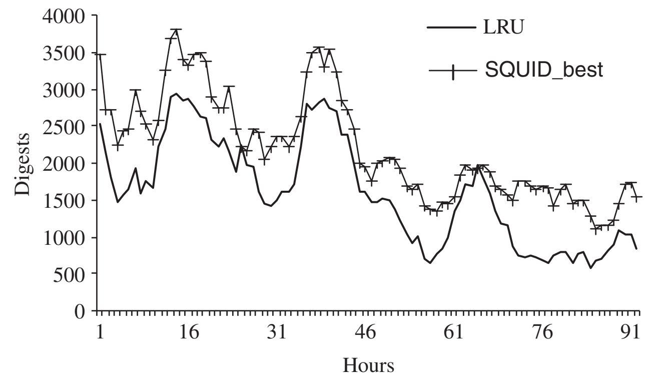

Fig. 10. The number of digests exchanged (SYNTH I).

applying the *All_to_all* method. Instead, we compare LRU (using the same values for parameters a, k and h) with the best Squid alternative obtained after executing 30 random configurations. Fig. 9 shows the results.

The improvement of LRU in this case is impressive. The small number of neighbors with respect to the total number of proxies restricts the benefit of static schemes, which can only search in their proximity. On the other hand, LRU, even with a limited number of exploration hops (2), can gradually relate nodes that are several hops apart through the intermediate proxies in their path.

Similar to Fig. 7, Fig. 10 compares the overhead of LRU and *Squid_best* in terms of the number of digest transfers. Since the network now contains 45 proxies (as opposed to 9 in the first experiment), the overhead of *Squid_best* is about 5 times higher. LRU is less sensitive to network size since the frequency of digest exchanges also depends on the quality of the neighbors. Another subtle point refers to the utilization of digests. According to Squid, a proxy will broadcast the new version of its digest to all its neighbors even if it is not useful to them. On the other hand, all our policies update digests on-demand; that is, new versions are only requested by neighbors that actually use them. If the proxy administrators need to further reduce the overhead, our methods can provide this option by keeping less updated versions of the cache digests. For instance, when a node v_i drops one of its current neighbors v_j, it can still keep its digest. Later on, if v_j becomes again a neighbor of v_i, v_i will not request an updated version of the digest unless the false hit ratio exceeds a certain threshold. Squid can also reduce the number of exchanges by increasing the update threshold from 1% to a higher percentage.

Finally, we tested the effect of the various parameters (*a*, *k* and *h*) on the performance of LRU. The results were expected and omitted here. In particular, the neighbor hits, and the rate of digest transfers increase with the number of neighbors (k) and the exploration probability (a). On the other hand, although the exploration messages increase exponentially with the maximum number of hops (h), the page hits and digest exchanges are not influenced considerably. This implies that if a page can be found in the network, it probably lies in the neighborhood of the requesting proxy and extensive exploration is not usually beneficial. We also replaced SYNTH I with SYNTH II and observed almost identical results to the ones in Figs. 9 and 10, suggesting that the performance of LRU depends on the total number of potential neighbor hits rather than the structure or configuration of individual proxies. In summary, LRU increases significantly the number of neighbor hits, especially for large networks. An obvious improvement over LRU concerns the reduction of digest transfers. Towards this direction, we evaluate the performance of LFU.

5.3. Unidirectional LFU vs. LRU

Here, we compare LRU and LFU. The same parameter values are used for both methods ($a = 0.5$, $k = 3$ and $h = 2$). Furthermore, l (number of positive responses required for reorganization) for LFU is set to 100, whereas r (weight factor for neighbor responses) is set to 0.5.

The first experiment in Fig. 11 shows the relative benefit of LFU for NLANR, SYNTH I and SYNTH II. The benefit is measured as *#LFU hits−#LRU hits* and can be positive or negative, depending on whether the number of hits increases or decreases. LRU is better for NLANR (and to a lesser extent for SYNTH I), while LFU is better for SYNTH II. The reason behind this trend is that both synthetic datasets are generated by randomly splitting the 9 original traces. The effect of this splitting is the disruption of the spatial locality of the traces. In other words, a page that should have been requested at a certain proxy due to the spatial locality of web requests is instead requested at some other proxy. The intuition behind LRU is to exploit this spatial locality. LFU, on the other hand, identifies proxies that are beneficial in the long run, regardless of the extent of spatial locality. Therefore, the performance of LRU degrades for SYNTH I, and is even worse for SYNTH II since the requests are distributed equally (and randomly) among all the proxies.

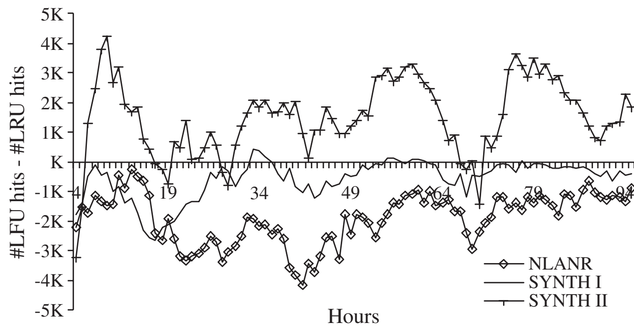

Fig. 11. The benefit of LFU in terms of neighbor hits.

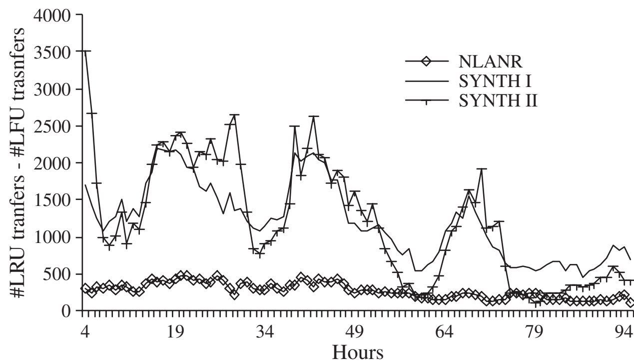

Fig. 12. The benefit of LFU in terms of digest exchanges.

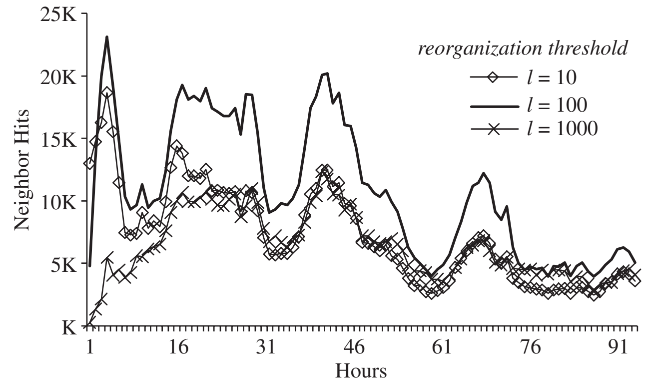

Fig. 13. The number of neighbor hits for various reorganization thresholds.

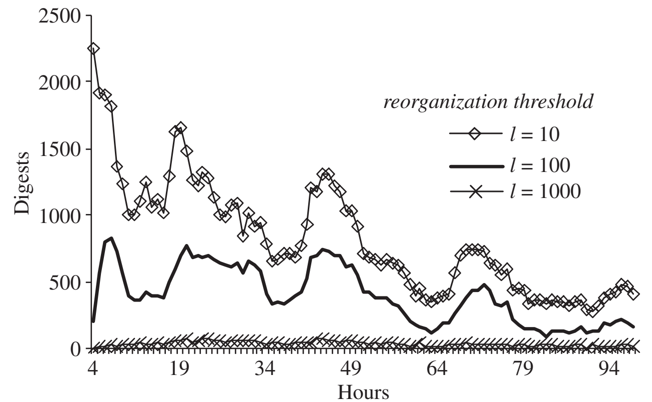

Fig. 14. The number of digests exchanged for various reorganization thresholds.

Next we measure the benefit of LFU in terms of digest transfers. Fig. 12 illustrates *#LRU transfers−#LFU transfers* for NLANR, SYNTH I and SYNTH II. The advantage of LFU is clear since it reduces considerably the network overhead in all cases. The difference is higher for the larger networks, indicating better scalability. A comparison with the absolute values of LRU for NLANR (Fig. 7) and SYNTH I (Fig. 10) suggests savings up to 70–80%.

The effects of the common parameters (*a, k* and *h*) are similar to LRU and not included. We only investigate the impact of the reorganization threshold (for $l = 10$, 100 and 1000) on SYNTH I. The number of hits (Fig. 13) is optimized for $l = 100$. If $l = 10$, LFU does not have enough statistics to select "good" neighbors, whereas if $l = 1000$, LFU cannot follow closely the changing request patterns. The network overhead caused by digest exchanges (Fig. 14) is inversely proportional to the value of l. In general, the proper tuning of l is crucial for achieving good performance, while maintaining low overhead. An optimal value of l is difficult to compute, since in addition to the traces, it depends on the proxy configuration and the values of the other LFU parameters.

In summary, LFU with appropriate parameter tuning is the best unidirectional method since it achieves a similar number of neighbor hits with LRU, with a significantly lower overhead.

5.4. Bidirectional strategies

The existence of symmetric relationships has some implications on the performance of the caching strategies as discussed in Section 4. One such effect is that the number of neighbor changes, and consequently of digest transfers, is expected to increase significantly due to the propagation of updates. In this section we compare the bidirectional alternatives using SYNTH I, where the number of neighbors for each proxy is uniformly distributed in the range 2–6. All neighborhood relations are symmetric.

Fig. 15 illustrates the number of neighbor hits obtained by B-LRU, B-LFU and Squid (Squid is also restricted to symmetric configurations). Although B-LRU is the best technique, as shown in Fig. 16, it incurs the highest overhead. B-LFU, on the other hand, is best in terms of digest transfers, while its performance is far superior to the static configuration.

In the next section we test B-LRU and B-LFU against their unidirectional counterparts in order to identify the differences in their behavior, and get an insight on the nature of content sharing that they achieve.

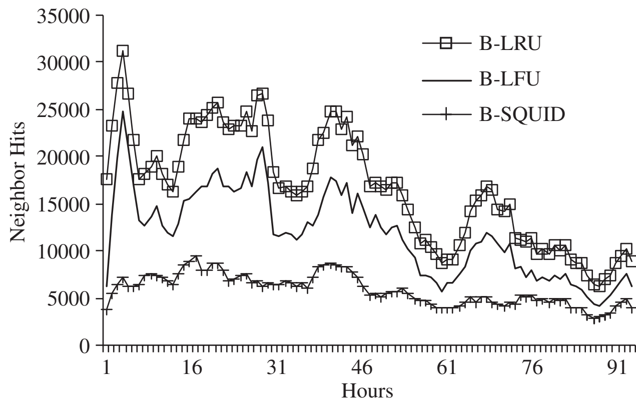

Fig. 15. The number of neighbor hits for bidirectional methods.

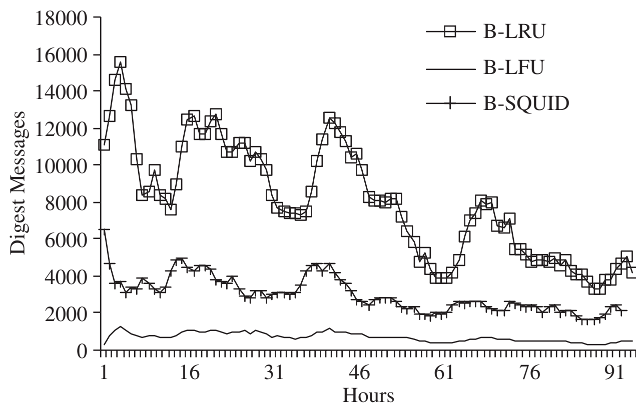

Fig. 16. The number of digest exchanges for bidirectional methods.

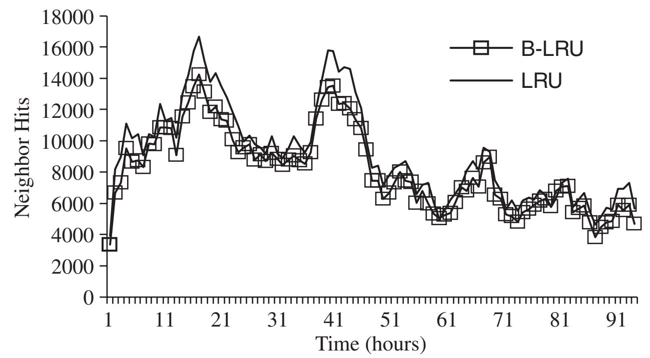

Fig. 17. The number of neighbor hits for LRU and B-LRU.

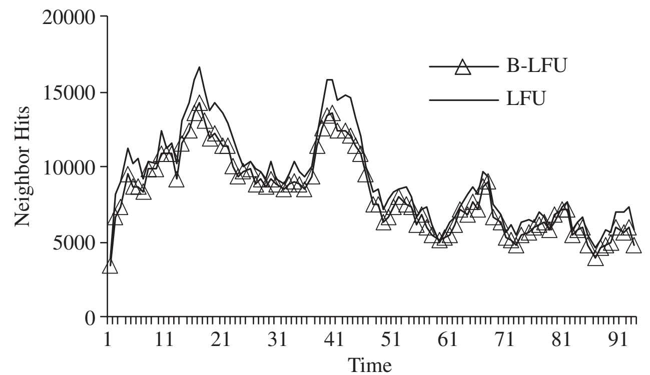

Fig. 18. The number of neighbor hits for LFU and B-LFU.

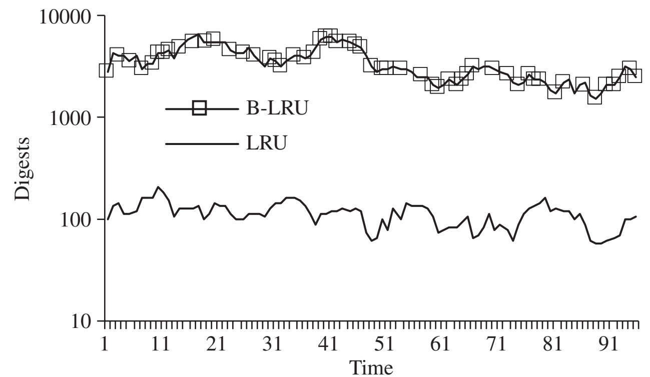

Fig. 19. The number of digest exchanges for LRU and B-LRU.

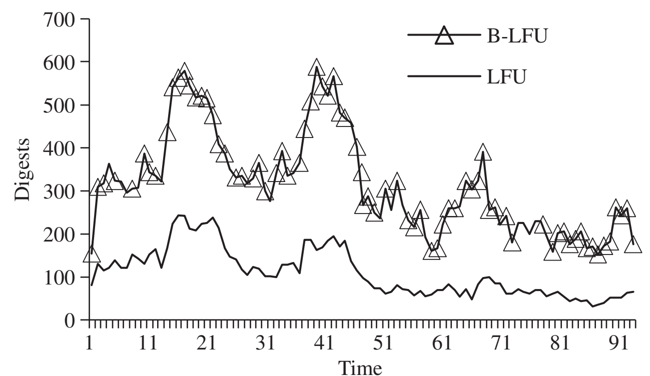

Fig. 20. The number of digest exchanges for LFU and B-LFU.

5.5. Sharing behavior

The first experiment compares uni- and bidirectional versions of LRU and LFU using the NLANR proxies in a configuration where each node has exactly four neighbors. As shown in Figs. 17 and 18, the two versions have very similar neighbor hits, for both LRU and LFU. This can be explained since, due to the large number of neighbors with respect to the network size, a node can easily locate its neighbors. Unidirectional schemes are slightly better due to the un-constrained selection.

The main difference, though, refers to the overhead in terms of digest transfers as shown in Figs. 19 and 20. B-LRU incurs between one and two orders of magnitude more transfers than LRU. On the other hand, the bidirectional version of LFU behaves relatively better since, as discussed in Section 5, it limits the propagation effect.

Finally, we explore the content sharing patterns imposed by the various alternatives. In particular, we choose one of the proxies (*bo*2) and illustrate in Fig. 21 the number of pages sent to or received from other proxies depending on the caching policy. Notice that the proxies on the x-axis are sorted according to their cache size (which is set to 10% of the total size of the locally requested objects). *bo*2 is the second smallest proxy after *startap*.

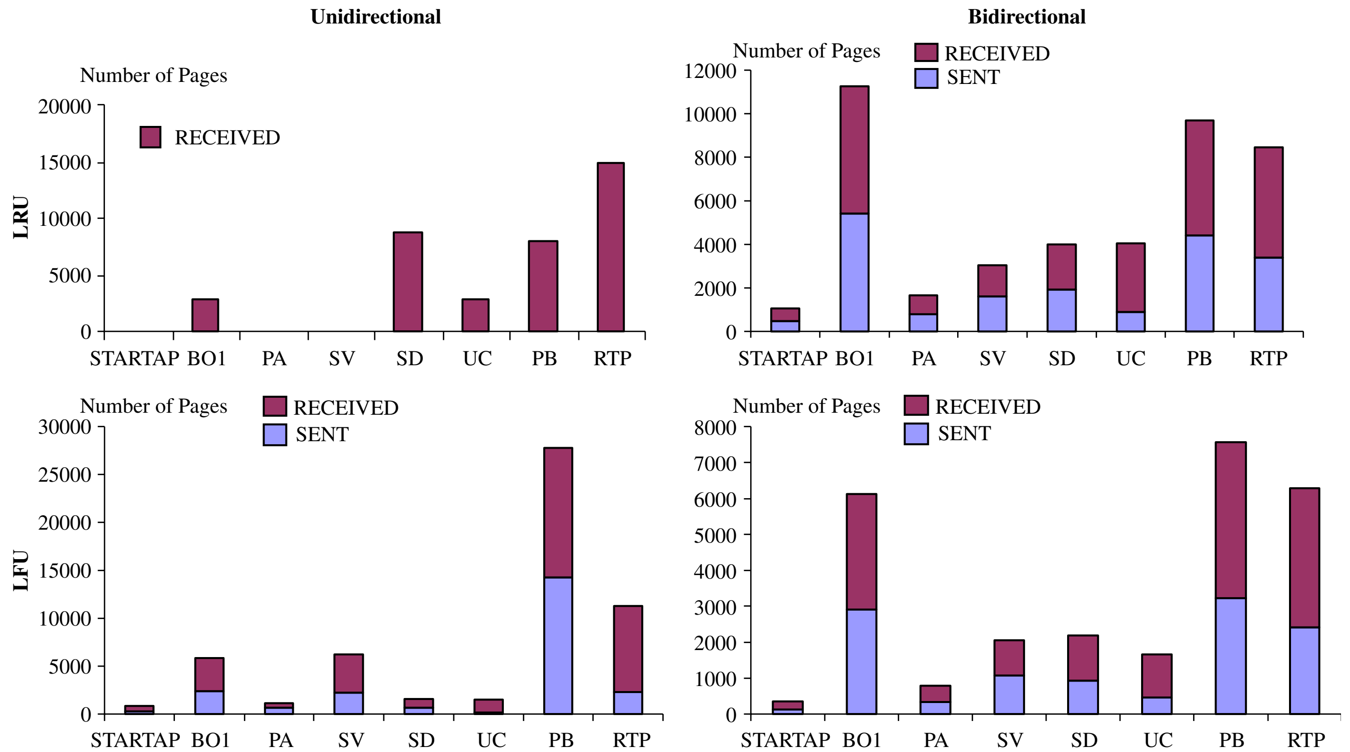

Fig. 21. Sharing patterns for *bo*2.

With unidirectional LRU (upper left diagram) *bo*2 only receives pages without servicing any requests. Moreover, most of these pages come from large proxies. This actually is a common pattern for all small proxies: they attach themselves to some large cache and remain there most of the time. In this case, the neighbors of *bo*2 are: *bo*1 (which as will see has very similar contents with *bo*2) and the four largest proxies in the network. This situation is not desirable since it may lead to over-congestion of the popular nodes.

LFU (lower left diagram) on the other hand, achieves some kind of load balancing since *bo*2 exchanges pages with all proxies. The explanation is that when *bo*2 joins the neighborhood of another proxy, it will remain there until the next reorganization phase. During this period it serves requests from the other proxy, thus the number of pages sent to other nodes is increased with respect to LRU.

The load balancing effect is even stronger for bidirectional LRU, since as soon as *bo*2 configures a neighbor, this proxy is enforced to query *bo*2. Although this approach may not be beneficial to some proxies (as suggested by the frequent neighbor changes observed in the previous experiments), it may help identify nodes with similar access patterns, which otherwise would be missed. A clear example of this situation is the level of content sharing between *bo*1 and *bo*2, which is very high given their small sizes. Similar observations hold for B-LFU.

5.6. Summary

The overall conclusion is that unidirectional strategies achieve the best performance in terms of both neighbor hits and traffic overhead. Specifically, unidirectional LRU is able to closely follow the changes in access patterns, by frequently changing the neighborhood list. It achieves a near-optimal hit ratio with a significantly smaller number of neighbors (i.e., lower cost). LFU, on the other hand, changes the neighborhood list periodically, based on the collection of statistics during the reconfiguration period. The result is a considerably lower amount of overhead traffic, and savings up to 80% (compared to LRU) were observed. Furthermore, LFU performs slightly better in terms of neighbor hits, when the selection of "good" neighbors is not very clear.

Bidirectional strategies enforce a symmetric relationship between neighbors. Therefore, they tend to limit the performance in terms of neighbor hits, since a proxy may not have an unlimited number of incoming neighbors. The benefit, though, is a load balancing effect, which is due to the symmetric relationships. Moreover, with bidirectional strategies the traffic overhead is increased significantly, because of the propagation effect of neighbor updates. While the amount of overhead is prohibitive for implementing a B-LRU strategy, the bidirectional version of LFU achieves a good balance between neighbor hits, traffic overhead, and load balancing.

6. Conclusions

In this paper we addressed two questions: Is it possible to improve the performance of current distributed web caching schemes using adaptive neighbor reconfiguration? If so, can

we tackle the resulting problem as a second level caching? Evidence from the simulation results provides a definitive yes to both the questions. LRU and LFU achieve higher hit ratios compared to their static counterparts in all experimental datasets. Even in small network instances, where an all-to-all neighbor configuration is feasible, adaptive caching techniques are useful as they achieve comparable performance at only a fraction of the bandwidth overhead.

Furthermore, the second level caching formulation provides a simple framework that permits the application of previous results to this problem. A straightforward extension of this work is to exploit other caching strategies that integrate latency, recency, frequency of requests, etc. Such techniques could be used to minimize measures like average response time or byte hit ratio.

An interesting alternative worth investigating is a centralized approach according to which a server keeps the digests of all participating proxies. Instead of issuing exploration queries, proxies would contact this server in order to locate potential neighbors. Although this method would probably decrease the exploration overhead, it suffers from the usual drawbacks of centralized techniques: (i) single point of failure, (ii) limited scalability, and (iii) performance bottleneck.

References

[B70] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.

[BCZ98] B. Bhattacharjee, K. Calvert, E. Zegura, Self-organizing wide area network caches, IEEE INFOCOM 1998.

[CI97] P. Cao, S. Irani, Cost-aware WWW proxy caching algorithms, USENIX Symposium on Internet Technology and Systems, 1997.

[CDN+96] A. Chankhunthod, P. Danzig, C. Neerdals, M. Schwartz, K. Worell, A hierarchical internet object cache, USENIX Technical Conference, San Diego, CA, 1996.

[DR01] S. Dykes, K. Robbins, A viability analysis of cooperative proxy caching, IEEE INFOCOM 2001.

[DRJ01] S. Dykes, K. Robbins, C. Jeffery, Uncacheable documents and cold starts in web proxy cache simulations: how two wrongs appear right, Technical Report CS-2001-01, University of Texas at San Antonio, Division of Computer Science, San Antonio, TX 78249-0664, January 2001.

[FCA+98] L. Fan, P. Cao, J. Almeida, A. Broder, Summary cache: a scalable wide-area web cache sharing protocol, ACM SIGCOMM 1998.

[GEC+99] M. Gullickson, C. Eiccholz, A. Chervenak, E. Zegura, Using experience to guide web server selection, Multimedia Computing and Networking, January 1999.

[KS98] P. Krishnan, B. Sugla, Utility of co-operating web proxy caches, Comput. Networks ISDN Systems 30 (1–7) (1998) 195–203.

[RCG98] M. Rabinovich, J. Chase, S. Gadde, Not all hits are created equal: cooperative proxy caching over a wide-area network, Third International Web Caching Workshop, 1998.

[R92] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, April 1992.

[RSB99] P. Rodriguez, C. Spanner, E. Biersack, Web caching architectures: hierarchical and distributed caching, Fourth International Web Caching Workshop, 1999.

[RW98] A. Rousskov, D. Wessels, Cache digest, Comput. Networks ISDN Systems 30 (22–23) (1998) 2155–2168.

[TDV+99] R. Tewari, M. Dahlin, H. Vin, J. Kay, Beyond hierarchies: design considerations for distributed caching on the internet, 19th International Conference on Distributed Computing Systems (ICDCS'99), June 1999.

[VR98] V. Valloppillil, K. Ross, Cache array routing protocol v1.0. IETF Internet Draft, February 1998, available at: http://www.globecom.net/ietf/draft/draft-vinod-carp-v1-03.html.

[W97] Z. Wang, Cachemesh: a distributed cache system for World Wide Web, Second International Web Caching Workshop, 1997.

[WC97] D. Wessels, K. Claffy, Internet cache protocol (ICP) version 2. RFC2186, September 1997.

[W02] C. Williamson, On filter effects in web caching hierarchies, ACM Trans. Internet Technol. 2 (1) (2002) 47–77.

[WVS+99] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, On the scale and performance of cooperative web proxy caching, 17th ACM Symposium on Operating System Principles (SOSP'99), 1999.

[WWW1] D. Wessels, Squid internet object cache, available at: http://www.squid-cache.org/.

[WWW2] National Lab of Applied Network Research, IRCache Project, Sanitized access logs, available at: http://www.ircache.net/.

[ZMN+97] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, V. Jacobson, Adaptive web caching: towards a new global caching architecture, Third International Web Caching Workshop, 1998.

Spiridon Bakiras received his B.S. degree (1993) in Electrical and Computer Engineering from the National Technical University of Athens, his M.S. degree (1994) in Telematics from the University of Surrey, and his Ph.D. degree (2000) in Electrical Engineering from the University of Southern California. Currently, he is a Postdoctoral Fellow in the Department of Computer Science at the Hong Kong University of Science and Technology. His research interests include high-speed networks, peer-to-peer systems, mobile computing, and spatial databases. He is a member of the ACM and the IEEE.

Dr. Thanasis Loukopoulos received his Diploma in Computer Engineering and Informatics from the University of Patras, Greece, in 1997. He was awarded a Ph.D. degree in Computer Science by the Hong Kong University of Science and Technology (HKUST) in 2002. After receiving his Ph.D. he worked as a Visiting Scholar in HKUST. Currently, he is a Visiting Lecturer at the Department of Computer & Communication Engineering of the University of Thessaly, Greece. His research interests include: Data management in Content Distribution Networks, Video Servers, P2P and Ad-Hoc Networks.

Dimitris Papadias is an associate professor at the Computer Science Department, Hong Kong University of Science and Technology. Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He has published extensively and been involved in the program committees of all major Database Conferences, including SIGMOD, VLDB and ICDE.

Ishfaq Ahmad received a B.Sc. degree in Electrical Engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1985, and a MS degree in Computer Engineering and a Ph.D. degree in Computer Science from Syracuse University, New York, U.S.A., in 1987 and 1992, respectively. His recent research focus has been on developing parallel programming tools, scheduling and mapping algorithms for scalable architectures, heterogeneous computing systems, distributed multimedia systems, video compression techniques, and web management. His research work in these areas is published in over 150 technical papers in refereed journals and conferences.

He is currently a full professor of computer science and engineering in the CSE Department of the University of Texas at Arlington. AT UTA, he leads IRIS (Institute for Research In Security), a multi-disciplinary research center engaged in safety and security related technologies. He is an associate editor of Cluster Computing, Journal of Parallel and Distributed Computing, IEEE Transactions on Circuits and Systems for Video Technology, IEEE Concurrency, and IEEE Distributed Systems Online.