Connection Caching

Edith Cohen *Haim Kaplan* *Uri Zwick*

AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932 USA
{edith,hkl}@research.att.com

Tel-Aviv University
Tel-Aviv 69978
Israel
zwick@math.tau.ac.il

Abstract

Communication between clients and servers in the Web is performed using TCP (Transmission Control Protocol). TCP first opens a connection between the two nodes and then sends data packets via this open connection. The time needed to open a connection contributes to the latency of servicing a request. Hence, latency can be reduced if open connections are kept idle and re-used for future requests. Indeed, new versions of HTTP support that. Web servers and proxies typically maintain many open connections, many of them idle. The number of connections each node can keep open is limited. Hence, connection management is increasingly perceived and handled as a caching problem.

A fundamental difference between *connection-caching* and standard caching is that connection caching involves multiple caches that affect each other. Each open connection is cached at two nodes. If one of these nodes decides to terminate the connection, the other node is forced to accept this decision.

We propose a theoretical model for connection caching and study the connection-caching problem in this model. While the ordinary off-line caching problem is easily solved in polynomial time, we show that the off-line connection caching problem is NP-complete, and in fact also APX-complete, i.e., it is even hard to approximate. We complement this result by showing that a distributed version of Belady's algorithm, which is optimal for a single cache [4], is a 2-approximation algorithm for off-line connection-caching. We provide examples that show that the analysis of this algorithm is tight.

We also show that any (on-line) c-competitive algorithm for ordinary caching can be converted into a distributed (on-line) $2c$-competitive algorithm for the connection-caching problem. The factor 2 here cannot be improved as follows from the preceding result. The result applies also to randomized algorithms.

As a consequence we get that a distributed version of LRU is $2k$-competitive, where k is the size of the largest cache in the network. By studying the distributed version of LRU directly, we show that it is in fact $(2k-1)$-competitive. The analysis here is again tight.

Finally, it follows from the general result stated above that a distributed version of the random marking algorithm is $O(\log k)$ competitive.

1 Introduction

Information exchange between clients and servers in the Web is performed using HTTP (Hyper Text Transfer Protocol). HTTP typically uses TCP (Transmission Control Protocol) as the underlying transport protocol. Hence, HTTP request and response messages are carried on a TCP connection established between the two entities.

TCP connection establishment involves a round-trip delay which induces latency on the user. Further delay is caused by "TCP slow start" which is essentially the several round-trip times that TCP uses until it "finds" the right transmission speed.

Each user-server session, however, involves multiple HTTP requests: Typical Web pages consist of an HTML page and multiple embedded images, each requested separately. Users are also likely to follow up with further requests to the same server. Early versions of HTTP closed the TCP connection after servicing each request. Studies indicate that connection-establishment is a significant component of request-response time [8] and that the short TCP connection associated with a single HTTP request rarely gets past the "slow start" phase [12]. In order to improve user-perceived latency, a case was made for *persistent HTTP* [18, 17]. A persistent TCP connection is kept open and re-used to carry imminent future HTTP requests. The common older HTTP/1.0 supports a *keep alive* option which instructs the server to keep the TCP connection open beyond the handling of the current request. With the newer HTTP/1.1, persistence is a default, and a connection is kept open, and idle, for a certain period of time until either a new request arrives or until the connection is closed by the client or the server.

Persistent connections imply that clients and servers keep many connections open, some for idle periods, in anticipation of future requests. Requests that utilize an already-open connection benefit from reduced latency. Recent study [8] indicates that connection-caching and re-use at proxy caches is more effective for reducing user-perceived latency than document caching. Since TCP is implemented on top of the connection-less IP, the overhead associated with keeping connection state information only exists at the two end-points. Open connections consume a socket and allocated memory for socket buffers, at each end. Since memory and sockets are limited, each entity – individual client, proxy, or Web server, has a limit on the number simultaneously open TCP connections.

With the deployment of HTTP/1.1, persistent TCP connections become cacheable re-usable resources. Indeed, HTTP/1.1 -compliant Web browsers, proxies, and servers (for example [2]) implement policies to manage TCP connections. Various policies for server-side connection management were proposed and studied at [7].

A fundamental difference between *connection-caching* and standard caching, is that connection-caching involves multiple caches that affect each other. Each open connection is cached by two entities. If one of the entities decides to terminate ("evict") the connection, the other entity is forced to accept this decision.

In this work, we present a theoretical model for connection-caching, and provide upper and lower bounds on the performance of various replacement policies.

We first establish that, in contrast to single-cache paging, the off-line problem of minimizing the number of requests encountering a closed connection ("misses") is NP-complete, and also APX-complete [3], i.e., there is no polynomial time approximation scheme (PTAS) for it, unless P=NP.

We then consider distributed connection-caching policies derived from common single-cache policies, such as LFD (Largest Future Distance, also known as Belady's algorithm) and LRU (Least Recently Used). In the distributed connection-caching set-up, each node runs a single-cache policy locally. We consider two versions of the connection-caching problem. In the first version, when one side decides to close (evict) a connection, the other side is notified instantly. In the second version, the other side realizes that a connection was closed only when a request for this connection arrives. For TCP connections, the first variant is more realistic, as TCP connections are established and terminated using a 3-way handshake. Thus, unless an end-point fails or communication is interrupted, nodes are notified when connections are terminated. Even though the first variant is more realistic, we also consider the second variant as it helps us analyze the first variant.

As we show in Section 2.2, any ordinary, single cache, caching algorithm A can be converted into two distributed connection-caching algorithms $\mathcal{A}$ and $\mathcal{A}^*$. The connection-caching algorithm $\mathcal{A}^*$ is obtained by running the caching algorithm A distributively at each node of the network, *without* taking into account notices of connection closures. The connection-caching algorithm $\mathcal{A}$ is obtained by running A at each node of the network while taking into account notices of connection closures. Algorithm $\mathcal{A}^*$ can be used therefore in both versions of the connection-caching model, while algorithm $\mathcal{A}$ can only be used in the version in which nodes are notified of connection closures.

It may seem that $\mathcal{A}^*$ is always weaker than $\mathcal{A}$, as $\mathcal{A}^*$ does not take all the information available to it into account. Algorithm $\mathcal{A}^*$, however, is much easier to analyze, and as we shall see, there are cases in which the performances of $\mathcal{A}$ and $\mathcal{A}^*$ are incomparable, i.e., there are sequences on which $\mathcal{A}$ behaves better and sequences on which $\mathcal{A}^*$ behaves better.

Utilizing the close connection between an ordinary caching algorithm A and its distributed counterpart $\mathcal{A}^*$, we show that if A is c-competitive, then $\mathcal{A}^*$ is $(2c)$-competitive. This holds whether A is an on-line or an off-line algorithm, and whether it is a deterministic or a randomized algorithm.

As LFD is an optimal single-cache caching algorithm (note that LFD is an off-line algorithm), we get that $\mathcal{LFD}^*$ is a 2-approximation algorithm for the connection-caching problem. As LRU is a k-competitive single-cache caching algorithm (note that LRU is an on-line algorithm), we get that $\mathcal{LRU}^*$ is a $(2k)$-competitive connection caching algorithm, where k is the size of the largest cache in the network. The last result can be generalized to show that $\mathcal{LRU}^*$ is $2k/(k-h+1)$-competitive versus off-line adversary if at each node it is allowed to use $k-h$ slots more than the adversary.

We further show that $\mathcal{LFD}$ is also a 2-approximation algorithm for connection-caching, and that $\mathcal{LRU}$ is a $(2k-1)$-competitive on-line algorithm for connection-caching, where k is again the size of the largest cache in the network. We provide matching lower bounds for the competitive ratio of $\mathcal{LFD}^*$, $\mathcal{LFD}$, and $\mathcal{LRU}$.

In practice, connections are managed by simple distributed online policies, such as LRU. The $2k/(k-h+1)$-competitiveness result for $\mathcal{LRU}^*$ and the $(2k-1)$-competitiveness result for $\mathcal{LRU}$ may be thought of as providing some theoretical support for this practice.

Related work Another context where connections are treated as a cacheable resource is maintaining virtual circuits for routing IP-traffic over ATM networks. This was studied extensively (see for example, [14, 15, 13]), but essentially, existing theoretical and experimental work treated it as a standard paging problem (considering a single node independently).

2 Model for connection caching

Let $G=(V,E)$ be an undirected graph representing a communication network. The vertices of V correspond to nodes in the network. The edges of E correspond to possible connections between them. Each possible connection $e\in E$ may be either *open* or *closed*. Initially all possible connections are closed. A vertex $v\in V$ may maintain at most $k(v)$ open connections at any given time. The parameter $k(v)$ is referred to as the size of the *connection cache* of v. The graph $H=(V,F)$ composed of all the open connections is therefore a subgraph of $G=(V,E)$ in which the degree of each vertex v is at most $k(v)$.

Initially all connections are closed. The network is then supposed to service a sequence $\sigma=\langle\sigma_1,\ldots,\sigma_\ell\rangle$ of *connection requests*. Each request $\sigma_i\in E$ is an edge in the graph $G=(V,E)$. If the connection σ_i is currently open, then the request can be serviced at no cost. If the connection $\sigma_i=(u,v)$ is currently closed, then it must be reopened to service the request. Opening a connection incurs a fixed *activation cost*. If u already maintains $k(u)$ open connections, then one of these connections must be closed before the connection $\sigma_i=(u,v)$ is open. The same is true for v. Opening a connection may thus force the closure of two other connections.

As mentioned in the introduction, the connection-caching model comes in two flavors. In the first flavor, nodes are immediately notified when connections are closed by the other side. In the second flavor, nodes do not get such notifications.

Our goal is to minimize the total activation cost incurred while servicing the sequence σ. We consider both on-line and off-line scenarios as well as distributed and centralized ones.

2.1 Varying establishment costs and buffer sizes

As in the case of a single-cache [10, 21, 22], the connection-caching model can be generalized by allowing varying sizes and fetching costs. In the TCP context, sizes may correspond to socket buffers allocations, but use of uniform sizes seems most reasonable. Establishment costs may capture the user-perceived latency caused by connection-establishment, and thus be asymmetric and vary according to connection-establishment-time or quality-of-service class. Varying costs seem appropriate for caching TCP connections.

2.2 Distributed algorithms

A *distributed* connection-caching algorithm, or policy, is an algorithm in which every node of the network runs a local replacement algorithm. The local replacement algorithm running in a specific node is only aware of connection requests that touch that node.

To serve the connection request $e = (u, v)$, the connection e must be in the caches of both u and v. If this is the case just before the request, then the request results in a *hit*, otherwise, it is a *miss*. If a request $e = (u, v)$ results in a miss, then both u and v are now forced to add e into their caches, if it is not already there. Suppose that $e = (u, v)$ is not currently in the cache of u. As u can only keep $k(u)$ connections in its cache, node u may be forced to *evict* a connection $e' = (u, v')$ from its cache. Similarly, if $e = (u, v)$ is not in the cache of v, then v may be forced to evict a connection from its cache. Consider now the fate of a connection $e' = (u, v')$ evicted from the cache of u. In the first variant of the connection-caching model, node v' is told that $e' = (u, v')$ was evicted from u's cache. The connection $e' = (u, v')$ may still be kept in the cache of v' but it is useless there. A request now for e' would still result in a miss, as e' in not in u's cache. We say, therefore, that the connection $e' = (u, v')$ is *dumped* from v's cache by its eviction from u. A dumped connection leaves a *hole* in the cache in which it is dumped from, i.e., a slot in the cache that does not hold a useful connection. In the second variant of the connection-caching model, connections are still dumped, and holes are still left, but nodes do not know that certain connections in their caches were dumped, until the next requests for these connections arrive.

Consider a single-cache replacement policy A. We can easily convert A into a distributed connection-caching algorithm $\mathcal{A}^*$ in the following way. Algorithm $\mathcal{A}^*$ instructs each node u to handle the caching of the connection at u using the policy A, *ignoring* dump notifications, if they arrive. Thus, for example, $\mathcal{LRU}^*$, the connection-caching algorithm obtained from the famous caching algorithm LRU behaves in the following way. If a connection has to be evicted from the cache of a node u, then the connection to be evicted would be the connection in u's cache whose last usage occurred before the usage of all the other connections in u's cache. In making this decision, $\mathcal{LRU}^*$ does *not* take into account any information, if it exists, that says that some of the connections in u's cache were dumped by their other endpoint and are therefore useless. Thus, $\mathcal{LRU}^*$ may evict an active connection from u's cache, even if u's cache contains holes.

In the second variant of the connection-caching model, in which nodes are not notified of connection closures made by the other side, the distributed connection-caching algorithm $\mathcal{A}^*$ seems to be the natural extension of the single-cache policy A. In the first model, in which nodes are told of dumps and know of holes in their caches, it seems wasteful not to use this information. We define, therefore, a second connection-caching algorithm $\mathcal{A}$, associated with the single-cache algorithm A. Algorithm $\mathcal{A}$ instructs each node u to handle its cache in the following way. If a connection e has to be added to u's cache, and u's connection cache contains at least one hole, then the connection e fills, i.e., replaces, one of these holes. (It does not matter which hole is used, if there is more than one.) If all the connections in u's cache are active, then algorithm A is used to decide which one of them to evict.

It may seem that $\mathcal{A}$ outperforms $\mathcal{A}^*$ on any request sequence. This, however, is not true, as we shall see in Section 7. Nevertheless, there are cases where we can relate the performance of $\mathcal{A}^*$ to the performance of $\mathcal{A}$. In the cases that we consider in this paper, the bounds that we get on the behavior of $\mathcal{A}^*$ are almost as strong as the bounds that we get for $\mathcal{A}$, even though $\mathcal{A}^*$ can be employed in a weaker model.

2.3 Connection-caching and ordinary caching

The connection-caching is easily seen to be a generalization of the ordinary caching problem. A caching problem with a cache of size k and with n pages can be cast as a connection-caching problem in a star network in which the center of the star has a cache of size k and each one of the n leaves of the star has a cache of size 1.

3 Off-line connection-caching is hard

We first prove that the centralized off-line connection-caching problem is NP-hard.

Theorem 3.1 *Finding the optimal way of serving a sequence σ of connection requests is an NP-hard problem.*

Proof. Let ϕ be a 3SAT instance. Let n be the number of variables appearing in ϕ and let m be the number of clauses in ϕ. We construct a graph $G = (V, E)$ and a sequence σ of connection requests such that σ can be served with $m + n$ hits if and only if the formula ϕ is satisfiable. (If the formula is not satisfiable, the number of hits would necessarily be smaller.)

The sequence σ is the concatenation of three sequences σ_1, σ_2 and σ_3 of connection requests. The sequences σ_1 and σ_3 correspond to the variables of ϕ.

While serving them, the connection-caching algorithm is effectively choosing a truth assignment to the variables appearing in ϕ. The sequence σ_2 corresponds to the clauses of ϕ. It is used to check whether the chosen truth assignment satisfies ϕ.

For each variable x appearing in ϕ we introduce a star with three leaves. Label the center x_1 and the leaves x, $\bar{x}$, and x_2. The capacity at the nodes x_1, x, and $\bar{x}$ is 2. The capacity of x_2 is 1. For each variable x appearing in ϕ, the sequence σ_1 contains the three requests

$$(x_1, x)\ ,\ (x_1, \bar{x})\ ,\ (x_1, x_2)\ ,$$

and the sequence σ_3 contains the two requests

$$(x_1, x)\ ,\ (x_1, \bar{x})\ .$$

The capacity of x_1 is 2 and the connection (x_1, x_2) is requested after requesting the connections (x_1, x) and $(x_1, \bar{x})$. As a consequence we get that after the network serves the requests in σ_1, the connection cache of x_1 contains at most one of the connections (x_1, x) and $(x_1, \bar{x})$. Caching (x_1, x) corresponds to setting the variable x to false, while caching $(x_1, \bar{x})$ corresponds to setting x to true. With any one of these choices, we score at most one hit while serving the requests (x_1, x) and $(x_1, \bar{x})$ of σ_2, a total of at most n hits for the whole of σ_2.

We now construct the sequence σ_2. For each clause C of ϕ, e.g., $C = x \vee \bar{y} \vee z$ we introduce 2 more vertices, C and $\hat{C}$. Vertex C has capacity 2 and $\hat{C}$ has capacity 1. For each clause C, the sequence σ_2 contains the following connection requests:

$$(C, x)\ ,\ (\hat{C}, x)\ ,\ (C, \bar{y})\ ,\ (\hat{C}, \bar{y})\ ,\ (C, z)\ ,\ (\hat{C}, z)\ ,$$
$$(C, \hat{C})\ ,$$
$$(C, x)\ ,\ (C, \bar{y})\ ,\ (C, z)\ .$$

As C has capacity 2, we can get at most one hit on each such subsequence. We can get that hit without dumping a connection of the form (x_1, x) if and only if at least one of the literals $x, \bar{y}$ or z is set to true. It is easy to verify that we get $m + n$ hits if and only if the formula is satisfiable. ■

A variation on the proof establishes that the connection-caching problem is hard to approximate.

Theorem 3.2 *The problems of minimizing the number of misses, or maximizing the number of hits that occur while servicing a sequence σ of connection requests are APX-hard.*

Proof. The reduction stays exactly as above. We view it this time, however, as a reduction from MAX 3SAT(3), the version of the unweighted MAX 3SAT problem in which each variable appears in at most 3 clauses and clauses are of size at most 3. It is known that MAX 3SAT(3) is APX-hard [19] (or MAX SNP-hard).

Let I be a instance of the MAX 3SAT(3) problem. We may assume that each variable x appears positively in two clauses, and negatively in another, or vice versa. (If all the appearances of x are positive or negative then it is clear which value we should assign to x.)

Let $\sigma = \sigma_1\sigma_2\sigma_3$ be the sequence of connection requests obtained from I using the reduction used in the proof of Theorem 3.1. We claim that it is possible to get at least $n + k$ hits while servicing this sequence if and only if it is possible to satisfy at least k clauses of the MAX 3SAT(3) instance I.

To prove the claim, we have to show that if it is possible to get at least $n + k$ hits, then it is possible to do so while getting n hits for the portion σ_3 of σ. This suffices, as the hits obtained while serving σ_3 determine the required truth assignment.

Suppose now that we do not get hits for (x_1, x) and $(x_1, \bar{x})$. As either x or $\bar{x}$ participate in only one clause, we get that either x or $\bar{x}$ participate in at most one hit. We can give up this hit in exchange for a hit on one of the requests (x_1, x) or $(x_1, \bar{x})$.

This shows that the problem of maximizing the number of hits is APX-hard. As in the instances of the connection caching problem generated by the above reduction, the maximal number of hits and the minimal number of misses are both at least a certain percentage of the number of requests, it follows that the problem of minimizing the number of misses is also APX-hard. ■

The off-line connection-caching problem with varying costs and sizes can be solved approximately in polynomial time, when buffer sizes and establishment costs of individual connections are small with respect to applicable cache sizes and the overall cost. The approximation is for the hit rate, and is obtained by generalizing the integer-program formulation and randomized-rounding application provided for an ordinary cache in [6].

We note that if the network is fixed, the connection-caching problem can be solved in linear time, in the length of the sequence, using dynamic programming.

4 Distributed connection-caching algorithms

In this section we begin to explore the behavior of distributed connection-caching algorithms derived from simple caching algorithms such as LRU and LFD. We show that $\mathcal{LFD}^*$ is a 2-approximation algorithm for the connection-caching problem and that $\mathcal{LRU}^*$ is an on-line $(2k)$-competitive algorithm for the connection caching problem, where k is the size of the largest cache in the network. Before proving these results, we introduce the notion of connection failures.

4.1 Misses versus failures

Let us review the connection-caching model we are using. According to our definition, a connection request $e = (u, v)$ results with a hit only if the connection e is currently open, i.e., if the connection e is cached by *both* u and v. A miss would occur even if the connection e is cached by one of u and v but not by both.

For accounting purposes, we introduce, in addition to misses, the notion of *failures*. We say that a connection request $e = (u, v)$ fails at u if $e = (u, v)$ is not in u's cache when the request arrives. Similarly, $e = (u, v)$ fails at v if $e = (u, v)$ is not in v's cache. (Note that we do not care here whether e is active at the time of the request. If the connection e was dumped by v but is still cached at u, then the request e would not fail at u.) Thus, each connection request may result in at most two failures. Each miss corresponds to either one or two failures.

We let $\mathrm{miss}(\mathcal{ALG},\sigma)$ and $\mathrm{fail}(\mathcal{ALG},\sigma)$ be the total number of misses, and failures, respectively, encountered by $\mathcal{ALG}$ while servicing the sequence σ. We let $\mathrm{miss}(\sigma)$ and $\mathrm{fail}(\sigma)$ be the minimal number of misses, or failures, respectively, needed to service σ. We clearly have:

Lemma 4.1 *For any sequence σ of connection requests:*

$$\mathrm{miss}(\sigma) \le \mathrm{fail}(\sigma) \le 2\cdot\mathrm{miss}(\sigma) .$$

Minimizing the total number of misses and minimizing the total number of failures are two related but different problems. We saw in the previous section that minimizing the number of misses is a difficult problem. Minimizing the number of failures, on the other hand, is an easy problem, at least in the off-line setting.

Theorem 4.2 *Let* OPT *be an optimal caching algorithm. Then the connection-caching algorithm $\mathcal{OPT}^*$ obtained from* OPT *services each connection request sequence with a minimal total number of failures.*

If establishment costs and buffer sizes are uniform, then LFD (Belady's algorithm) is an optimal caching algorithm ([4], see also [5]) and the connection-caching algorithm $\mathcal{LFD}^*$ therefore minimizes the total number of failures.

Proof. We let $\mathrm{fail}_v(\mathcal{ALG},\sigma)$ be the number of failures that occur at the node v while servicing σ using the connection caching algorithm $\mathcal{ALG}$. We clearly have:

$$\mathrm{fail}(\mathcal{ALG},\sigma) = \sum_v \mathrm{fail}_v(\mathcal{ALG},\sigma) .$$

The number of failures that occur at a node v of the network depend only on the caching decisions made at v. Dumps made by other vertices have no effect on the number of failures occurring at v. Thus, to minimize the number of failures at v we should manage v's cache using an optimal single-cache policy. This is exactly what is done by $\mathcal{OPT}^*$. ∎

As an immediate corollary of Theorem 4.2 and Lemma 4.1, we get the following:

Corollary 4.3 *The connection caching algorithm $\mathcal{LFD}^*$ is a 2-approximation algorithm for the connection-caching problem.*

4.2 Upper bounds on competitive ratio

A caching algorithm A is said to be c-*competitive* if for every sequence σ of requests we have $\mathrm{miss}(A,\sigma) \le c\cdot\mathrm{miss}(\sigma)$. (Sometimes, A is said to be c-competitive also if for every σ we have $\mathrm{miss}(A,\sigma) \le c\cdot\mathrm{miss}(\sigma)+d$, for some fixed constant d.) An identical definition applies to connection-caching algorithms.

Another simple consequence of Theorem 4.2 and Lemma 4.1 is the following result:

Theorem 4.4 *Let A be a c-competitive caching algorithm. Then, $\mathcal{A}^*$, the connection-caching algorithm obtained by running A distributively at each node of the network, ignoring notifications of connection closures, is a $(2c)$-competitive algorithm for the connection-caching problem.*

Proof. Let σ be a sequence of connection requests. As $\mathcal{A}^*$ runs A distributively at each node, and as A is c-competitive, we get:

$$\begin{aligned}\mathrm{miss}(\mathcal{A}^*,\sigma) &\le \mathrm{fail}(\mathcal{A}^*,\sigma) = \textstyle\sum_v \mathrm{fail}_v(\mathcal{A}^*,\sigma)\\ &\le \textstyle\sum_v c\cdot\mathrm{fail}_v(\sigma) = c\cdot\mathrm{fail}(\sigma) \le 2c\cdot\mathrm{miss}(\sigma) ,\end{aligned}$$

as required. ∎

Two remarks here are in order. First, if A is c-competitive with an additional additive term, then $\mathcal{A}^*$ would also have an additive term and this term would depend on the size of the network. Second, if, as is usually the case, the competitiveness ratio of A depends on the size of the cache, i.e., if A is $c(k)$-competitive for caches of size k, then $\mathcal{A}^*$ would be $(2c(k))$-competitive, where k is the size of the largest cache in the network.

A caching algorithm A is said to be $c(h,k)$-competitive, if for every sequence of requests, the number of misses incurred while servicing the sequence using a cache of size k managed by algorithm A is at most $c(h,k)$ times the minimal number of misses required to service the sequence with a cache of size h, plus perhaps a fixed constant. In the previous definition of competitiveness, h was taken to be k. As we are comparing here the behavior of an on-line algorithm with the optimal off-line algorithm, we sometimes want to weaken the power the off-line algorithm. We can do that by taking $h<k$.

Let G be a network. For every node v in the network, we let $k(v)$ be the size of the local connection-cache held at v. We compare the behavior of distributed connection-caching algorithms with the behavior of the optimal centralized off-line connection-caching algorithm which is allowed to keep a connection-cache of size $h(v)$ at each node v of the network. Theorem 4.4 can be easily generalized into the following theorem:

Theorem 4.5 *Let A be a $c(h,k)$-competitive caching algorithm. Let $\mathcal{A}^*$ be the distributed connection-caching algorithm obtained by running A locally at each node of the network, ignoring notifications of connection closures. Then, $\mathcal{A}^*$ is $2\cdot\max_v c(h(v),k(v))$-competitive.*

LRU (Least Recently Used) is the cache replacement policy which evicts the page that was requested least recently. It is easy to see that LRU, with a cache of size k, is k-competitive ([20], see [5] and [11]). Furthermore, LRU, with a cache of size k, is $k/(k-h+1)$-competitive versus an adversary that uses a cache of size h. As corollaries of Theorems 4.4 and 4.5, we get:

Corollary 4.6 *(i) $\mathcal{LRU}^*$ is a $(2k)$-competitive connection-caching algorithm, where k is the size of the largest cache in the network.*

(ii) $\mathcal{LRU}^*$ is $(2\max_v \frac{k(v)}{k(v)-h(v)+1})$-competitive connection-caching algorithm versus an adversary that keeps a cache of size $h(v)$ at each node v of the network.

Theorems 4.4 and 4.5 relate the performance of $\mathcal{A}^*$ to that of A. They do not directly relate the performance of $\mathcal{A}$ to that of A. In the case of LRU, we can directly relate the performance of $\mathcal{LRU}$ to that of $\mathcal{LRU}^*$.

Lemma 4.7 *For any sequence σ of connection requests we have* $\text{miss}(\mathcal{LRU}, \sigma) \leq \text{miss}(\mathcal{LRU}^*, \sigma)$.

Proof. (sketch) We show that the set of open connections held by $\mathcal{LRU}^*$, i.e., connections that are cached by both their end-points, is always a subset of the open connections held by $\mathcal{LRU}$. The details will be given in the full version of the paper. ■

As a consequence, we get that the results of Corollary 4.6 apply to $\mathcal{LRU}$ as well as to $\mathcal{LRU}^*$. A better bound on the ratio of $\mathcal{LRU}$ is given in Section 6.

There are randomized paging algorithms that are $O(\log k)$-competitive [16, 9, 1] (see also [5],[11]). As a consequence we get:

Corollary 4.8 *There are distributed randomized on-line $O(\log k)$-competitive connection caching algorithms.*

4.3 Lower bounds on competitive ratio

As mentioned in Section 2.3, connection-caching is a generalization of the classical single-cache paging model. It is known (see [5] and [11]) that the competitive ratio of any *deterministic* caching algorithm is at least k, where k is the size of the cache, and that the competitive ratio of any *randomized* caching algorithm is at least $H_k = \sum_{i=1}^{k} \frac{1}{k} > \ln k$ ([9], see also [5] and [11]). The same lower bounds apply, therefore, also to the connection-caching problem.

5 The performance of $\mathcal{LFD}$

Corollary 4.3 says that $\mathcal{LFD}^*$ is a 2-approximation algorithm for the connection-caching problem. What about $\mathcal{LFD}$? We have examples that show that there are sequences on which $\mathcal{LFD}$ behaves better $\mathcal{LFD}^*$, while there are other sequences on which $\mathcal{LFD}^*$ behaves better than $\mathcal{LFD}$. Nevertheless, we can show that $\mathcal{LFD}$ is also a 2-approximation algorithm for the connection-caching problem.

We let $\text{dump}_v(\mathcal{A}, \sigma)$ be the number of connections that are dumped from the cache of v, by v's neighbors, while servicing the request sequence σ using the connection-caching algorithm $\mathcal{A}$. We let $\text{dump}(\mathcal{A}, \sigma) = \sum_v \text{dump}_v(\mathcal{A}, \sigma)$ be the total number of dumps that occur while servicing σ using $\mathcal{A}$. Recall that $\text{miss}(\mathcal{A}, \sigma)$ and $\text{fail}(\mathcal{A}, \sigma)$ are the total number of misses and failures that occur while servicing σ using $\mathcal{A}$. We need the following simple Lemma:

Lemma 5.1 *For any connection-caching algorithm $\mathcal{A}$ and any request sequence σ we have* $\text{dump}(\mathcal{A}, \sigma) \leq \text{miss}(\mathcal{A}, \sigma)$.

Proof. Charge a half of each dump to the miss that caused it and the second half to the miss that would fill the hole caused by the dump. The second half of the last dump that occurs in each slot of a cache is charged to the miss that caused this slot to be used for the first time. In this way, each miss is charged for at most one dump. ■

We now claim:

Theorem 5.2 *$\mathcal{LFD}$ is a 2-approximation algorithm for the connection-caching problem.*

Proof. We let $\mathcal{A}$ denote any connection-caching algorithm. We let $\mathcal{B}$ stand for $\mathcal{LFD}$. We show that for every sequence σ and for every node v of the network we have:

$$\text{miss}_v(\mathcal{B}, \sigma) \;\leq\; \text{fail}_v(\mathcal{A}, \sigma) + \text{dump}_v(\mathcal{B}, \sigma)\,. \qquad (*)$$

Here $\text{miss}_v(\mathcal{B}, \sigma)$ is the total number of misses in requests that touch v incurred by $\mathcal{B}$ while servicing σ. By summing this inequality over all nodes we get

$$2 \cdot \text{miss}(\mathcal{B}, \sigma) \;\leq\; \text{fail}(\mathcal{A}, \sigma) + \text{dump}(\mathcal{B}, \sigma)\,.$$

By Lemma 5.1 $\text{dump}(\mathcal{B}, \sigma) \leq \text{miss}(\mathcal{B}, \sigma)$ and therefore $\text{miss}(\mathcal{B}, \sigma) \leq \text{fail}(\mathcal{A}, \sigma)$. As $\text{fail}(\mathcal{A}, \sigma) \leq 2 \cdot \text{miss}(\mathcal{A}, \sigma)$, we get that

$$\text{miss}(\mathcal{B}, \sigma) \leq 2 \cdot \text{miss}(\mathcal{A}, \sigma)\,.$$

As this holds for any connection-caching algorithm, including the optimal one, it follows that $\mathcal{B} = \mathcal{LFD}$ is a 2-approximation algorithm.

Before proving $(*)$ we introduce some new notation. For brevity, we omit σ from now on from the expressions $\text{miss}_v(\mathcal{B}, \sigma)$, $\text{dump}_v(\mathcal{B}, \sigma)$ and $\text{fail}_v(\mathcal{A}, \sigma)$. We let $\text{miss}_v(\mathcal{B})$, $\text{dump}_v(\mathcal{B})$ and $\text{fail}_v(\mathcal{A})$ be the number of misses, dumps and failures seen *so far* at node v. We also use $\text{cache}_v(\mathcal{A})$ and $\text{cache}_v(\mathcal{B})$ to denote the states of v's cache under $\mathcal{A}$ and under $\mathcal{B}$ after servicing a portion of the sequence σ. (To be more formal we would have to add an index of time to all these notations. We avoid that as this would make the notations very cumbersome.)

Consider a location a in $\text{cache}_v(\mathcal{A})$ and a location b in $\text{cache}_v(\mathcal{B})$. Let $e(a)$ and $e(b)$ be the connections stored in a and b. (The locations a and b may be empty, in which case the connections $e(a)$ and $e(b)$ are dummy connections.) We say that b *dominates* a, and write $b \geq a$, if either a is empty, or if $e(b)$ is *active* and either $e(a) = e(b)$, or the next request for $e(b)$ will occur before the next request for $e(a)$. (Note that this definition treats the connections in $\text{cache}_v(\mathcal{A})$ and $\text{cache}_v(\mathcal{B})$ in a different manner. If a is not empty, then for b to dominate a, the connection $e(b)$ must be active even if the connection $e(a)$ is not active.)

We define the *advantage* that $\text{cache}_v(\mathcal{A})$ has over $\text{cache}_v(\mathcal{B})$, denoted $\text{adv}_v(\mathcal{A}, \mathcal{B})$, as the minimal $c \geq 0$ such that there is a matching between the locations of $\text{cache}_v(\mathcal{A})$ and those of $\text{cache}_v(\mathcal{B})$ such that at most c of the locations in $\text{cache}_v(\mathcal{A})$ are *not* dominated by their mates in $\text{cache}_v(\mathcal{B})$.

We now prove the following strengthening of $(*)$. At every node of the network, and at any time during the servicing of σ we have:

$$\text{miss}_v(\mathcal{B}) + \text{adv}_v(\mathcal{A}, \mathcal{B}) \;\leq\; \text{fail}_v(\mathcal{A}) + \text{dump}_v(\mathcal{B})\,. \quad (**)$$

This is clearly a strengthening of $(*)$ as we always have $\text{adv}_v(\mathcal{A}, \mathcal{B}) \geq 0$.

We prove $(**)$ by induction. The claim clearly holds before servicing the first request. Suppose that $e = (v, w)$ is the next request to be serviced by $\mathcal{A}$ and $\mathcal{B}$.

Servicing $e = (v, w)$ by $\mathcal{B}$ may affect the states of at most four caches in the network. The caches $\text{cache}_v(\mathcal{B})$

and $\text{cache}_w(\mathcal{B})$ can clearly change as a result of this request. Furthermore, if e results with a miss, then v may be forced to dump a connection (v, v'), thereby changing $\text{cache}_{v'}(\mathcal{B})$, and w may be forced to dump a connection (w, w'), thereby changing $\text{cache}_{w'}(\mathcal{B})$ (v' and w' are not necessarily distinct).

Similarly, the servicing of $e = (v, w)$ by $\mathcal{A}$ may clearly change $\text{cache}_v(\mathcal{A})$ and $\text{cache}_w(\mathcal{A})$. Again, v and w may be forced to dump connections (v, v'') and (w, w''). But, as we do not distinguish between active and non-active connections in $\text{cache}_{v''}(\mathcal{A})$ and $\text{cache}_{w''}(\mathcal{A})$, we can consider these two caches to be unchanged.

We have to show that $(**)$ continues to hold, after the servicing of e by $\mathcal{A}$ and $\mathcal{B}$, at all the nodes of the network. If $u \notin \{v, w, v', w'\}$, then $(**)$ clearly continues to hold at u. Consider now the situation at v'; $\text{miss}_{v'}(\mathcal{B})$ and $\text{fail}_v(\mathcal{A})$ remain unchanged while $\text{adv}_{v'}(\mathcal{A}, \mathcal{B})$ and $\text{dump}_{v'}(\mathcal{B})$ may increase. However, as each connection dumped from $\text{cache}_{v'}(\mathcal{B})$ increases $\text{adv}_{v'}(\mathcal{A}, \mathcal{B})$ by at most one, as at most one domination is destroyed, $(**)$ continues to hold. The same applies to w''.

All that remains, therefore, is to show that $(**)$ continues to hold at v and w. We consider v. The same argument would apply of course also to w. We consider four different cases:

- **Case 1.** $\mathcal{B}$ gets a hit, $\mathcal{A}$ does not fail at v.
  We may assume that (e, e) is one of the good pairs is the matching that determines the value of $\text{adv}_v(\mathcal{A}, \mathcal{B})$. The pair (e, e) clearly remains good after servicing c. Thus, none of the quantities $\text{miss}_v(\mathcal{B})$, $\text{adv}_v(\mathcal{A}, \mathcal{B})$, $\text{fail}_v(\mathcal{A})$ and $\text{dump}_v(\mathcal{B})$ change as a result of the request and $(**)$ continues to hold.

- **Case 2.** $\mathcal{B}$ gets a miss, $\mathcal{A}$ fails at v.
  The quantities $\text{miss}_v(\mathcal{B})$ and $\text{fail}_v(\mathcal{A})$ both increase by one while $\text{dump}_v(\mathcal{B})$ stays unchanged. We have to show that $\text{adv}_v(\mathcal{A}, \mathcal{B})$ does not increase. The request for e forces $\mathcal{A}$ and $\mathcal{B}$ to add e to $\text{cache}_v(\mathcal{A})$ and $\text{cache}_v(\mathcal{B})$. Let a and b be the locations in $\text{cache}_v(\mathcal{A})$ and in $\text{cache}_v(\mathcal{B})$ in which e is placed. Let a' and b' be the locations matched with a and b. If at most one of these pairs was good, then $\text{adv}_v(\mathcal{A}, \mathcal{B})$ does not increase as the dominated pair that might have been lost can now be replaced by the dominated pair $b \geq a$. Assume, therefore, that before the request we had $a' \geq a$ and $b \geq b'$. As e was placed in b, and as $\mathcal{B}$ evicts the connection with the largest future distance, the next request of $e(b)$ must be after the next request of $e(a')$. Thus, we also have $a' \geq b'$. This would still hold after the request. After the request, we also have $b \geq a$, as $e(b) = e(a) = e$ and e is active. Thus, the number of dominated pairs does not decrease and $\text{adv}_v(\mathcal{A}, \mathcal{B})$ does not increase.

- **Case 3.** $\mathcal{B}$ gets a miss, $\mathcal{A}$ does not fail at v.
  We have to show that $\text{adv}_v(\mathcal{A}, \mathcal{B})$ decreases by at least one. As e is already cached at $\text{cache}_v(\mathcal{A})$ before the request for e, algorithm $\mathcal{A}$ has no reason to evict a connection from $\text{cache}_v(\mathcal{A})$. Let a be location of e in $\text{cache}_v(\mathcal{A})$. Algorithm $\mathcal{B}$, on the other hand, may be forced to evict a connection. Let b be the location in $\text{cache}_v(\mathcal{B})$ into which e is placed. Before the request, $e = e(a)$ is the request in $\text{cache}_v(\mathcal{A})$ with the nearest next request time, and $e(b)$ is either inactive or it has the latest request time among the active connections in $\text{cache}_v(\mathcal{B})$. Let a' and b' be the locations matched with a and b. It is not difficult to see that the value of $\text{adv}_v(\mathcal{A}, \mathcal{B})$, before the request for e, can be realized using a matching in which $a' \not\geq a$ and $b \not\geq b'$. After the request for e we have $b \geq a$ so $\text{adv}_v(\mathcal{A}, \mathcal{B})$ decreases by at least one.

- **Case 4.** $\mathcal{B}$ gets a hit, $\mathcal{A}$ fails at v.
  We have to show that $\text{adv}_v(\mathcal{A}, \mathcal{B})$ increases by at most one. Let b be the location of $\text{cache}_v(\mathcal{B})$ containing e. Let a be the location of $\text{cache}_v(\mathcal{A})$ into which e is placed. Let a' and b' be the mates of a and b. After the request we can match a with b and the number of good pairs decreases by at most one.

This completes the proof of $(**)$ and of the Theorem.

■

In Section 7 we provide an example that shows that the factor 2 for $\mathcal{LFD}$ is tight.

6 The performance of $\mathcal{LRU}$

We provide matching lower and upper bounds of $2k - 1$ on the competitive ratio of $\mathcal{LRU}$.

6.1 Lower bound

We construct a set of networks and corresponding request sequences on which $\mathcal{LRU}$ incurs $2k-1$ times the number of misses incurred by another (the optimal) algorithm.

We start with the $k = 2$ case on a graph with 6 nodes[1] and show a sequence where $\mathcal{LRU}$ incurs 3 times as many misses as $\mathcal{OPT}$.

The initial cache configuration is identical for both $\mathcal{LRU}$ and $\mathcal{OPT}$, and consists of two triangles with interleaving relative ages of their edges. Note that any initial cache configuration can be obtained with an appropriate sequence of requests.

We describe a sequence of 12 requests on which $\mathcal{OPT}$ incurs 4 misses while $\mathcal{LRU}$ incurs 12 misses. The sequence brings us back to a configuration isomorphic to the starting configuration. The sequence is illustrated in Table 1. Edges in $\mathcal{LRU}$'s cache are labeled with their birth time-stamp. The initial cache configuration consists of two disjoint "triangles," (a, b, f) and (c, d, e), Both algorithms incur misses on the first two requests: $\mathcal{LRU}$ evicts 2 edges on each request (4 in total) while $\mathcal{OPT}$ evicts 2 edges on the first request and none on the second (2 in total). Consequently, there are 4 edges that are cached by $\mathcal{OPT}$ and not by $\mathcal{LRU}$. These edges are then requested (4 misses for $\mathcal{LRU}$ and 4 hits for $\mathcal{OPT}$), bringing both caches to an identical 6-cycle configuration (a, e, c, d, b, f). To allow for a clearer illustration of requests 6–12 in Table 1, we exchanged the positions of

[1]This is the smallest number of nodes for which we can obtain a competitive ratio of 3 with $k = 2$.

nodes b and e. Another sequence of 6 requests, where both algorithms incur misses on the first 2 requests, and only $\mathcal{LRU}$ incurs misses on the last 4, bring both caches back to a configuration isomorphic to the starting configuration.

The state of $\mathcal{LRU}$'s cache is captured by the current cache configuration and a partial order providing the relative "age" of all edges with common endpoints. The following lemma was not needed for the $k = 2$ case but is used for the generalization.

Lemma 6.1 *If $\mathcal{LRU}$ and $\mathcal{OPT}$ are in the same cache configuration, the relative ages of the edges can be reordered arbitrarily without incurring any misses by either algorithm.*

Proof. We issue a series of requests that are hits for both $\mathcal{LRU}$ and $\mathcal{OPT}$. We request all edges one by one, starting from what we want to be the "oldest" edge to what we want to become the "newest" edge. ■

We now generalize the construction above to $k > 2$. The construction relies on the $k = 2$ case and contains the hexagon subgraph and the 12-requests sequence (broken to 4 subsequences with new requests inserted).

In addition, we use $k - 2$ sets of new edges (possible requests) $R_1, \ldots, R_{k-2}$. The R_i's are mutually disjoint. Each R_i contains 6 edges, and there is one edge incident to a hexagon node (a–f) and the other side incident to a new node. The new nodes are such that no more than k edges from $R_1 \cup \cdots \cup R_{k-2}$ are incident to them.[2] Note that this implies that all evictions are initiated at the hexagon nodes (a–f).

The construction is such that $\mathcal{OPT}$ never evicts R_i's edges. Each 6-sequence-part of the original 12-sequence is replaced by a new sequence of length $6 + 4(k - 2)$. The idea is that before evicting the 4 "hexagon" edges, $\mathcal{LRU}$ first evicts 4 R_1 edges, then these 4 edges are requested and $\mathcal{LRU}$ evicts 4 R_2 edges to accommodate them, and so forth until 4 R_{k-2} edges are evicted and requested. Only then, $\mathcal{LRU}$ evicts the 4 "hexagon" edges evicted in the $k = 2$ construction. In this process, $\mathcal{OPT}$ incurs only 2 misses while $\mathcal{LRU}$ incurs $4k - 2$ misses. A similar expansion is performed on the second part of the 12-sequence. In total, $\mathcal{OPT}$ incurs 4 misses while $\mathcal{LRU}$ misses on all $8k - 4$ requests, establishing that the competitive-ratio is at least $2k - 1$.

In the start of the first and second part (when $\mathcal{LRU}$ and $\mathcal{OPT}$ have the same configuration), we set the relative ages of edges from young to old to be such that the relative ages of the hexagon edges is the same as for the $k = 2$ case, all R_i edges are "older," and age is decreasing with index. For example, the corresponding order at the start of the first part is

$$\{(a,b),(d,e)\} < \{(b,f),(c,e)\} < \{(a,f),(c,d)\} < R_{k-2} < \cdots < R_1 .$$

The extended first part of the sequence (augmenting the first 6 requests) is provided in Table 2. The extended second part is similar. Note that at the end of each part, $\mathcal{LRU}$ and $\mathcal{OPT}$ have identical configurations, and the configuration at the end of the second part is isomorphic to the starting point.

[2]The construction requires $12 - \lceil 12/k \rceil$ nodes in total.

6.2 Upper bound

Theorem 6.2 *$\mathcal{LRU}$ is $(2k - 1)$-competitive.*

Our proof uses a potential function argument. The potential depends on the sequence and the current contents of $\mathcal{LRU}$'s and $\mathcal{OPT}$'s caches. Our potential function is

$$\Phi = C + (k - \frac{1}{2})H ,$$

where H is the current number of holes in $\mathcal{OPT}$'s cache and C is the number of edges (u, v) that satisfy the following two conditions:

1. (u, v) is currently cached by both $\mathcal{LRU}$ and $\mathcal{OPT}$.
2. Every edge e that (i) touches u or v, (ii) is cached by $\mathcal{LRU}$, and (iii) is not cached by $\mathcal{OPT}$, is older than (u, v).

We consider the effect of each request on the potential function. For accounting purposes, we treat the service of each request in two stages. In the first stage the request is serviced by $\mathcal{OPT}$ and in the second stage the request is serviced by $\mathcal{LRU}$. Note that requests that are hits do not change the potential. Also note that Φ is bounded by the size of the network and cache sizes. The following two lemmas quantify the effect of misses and conclude the proof of Theorem 6.2. Since Φ is maximized when the caches of $\mathcal{LRU}$ and $\mathcal{OPT}$ are both empty we obtain that $\mathcal{LRU}$ is $(2k-1)$-competitive with no additive term.

Lemma 6.3 *A miss of $\mathcal{OPT}$ decreases Φ by at most $2k - 1$.*

Proof. If the requested edge $e = (u, v)$ is cached by $\mathcal{LRU}$, then C is increased by 1. We now separately account for changes in Φ due to changes in the number of holes and evictions. We consider 3 cases:

1. Both caches of u and v had holes. In this case, H is decreased by 2 and C remains unchanged. In total, Φ decreases by $2k - 1$.
2. Only one of u and v (suppose u without loss of generality) had a hole. In this case, the hole at u is filled and another edge (v, w) is evicted at v, causing a new hole to be generated at w. Hence, the total number of holes remains unchanged. If the evicted edge (v, w) is cached by $\mathcal{LRU}$, it may disqualify other edges touching v and w from being accounted in C. In total, $k - 1$ edges touching each of v and w may be disqualified. These edges together with (v, w) may decrease C (and hence Φ) by at most $2k - 1$.
3. Neither u and v had holes. In this case, two edges are evicted (u, w) and (v, t). Two new holes are generated at w and t, increasing H by 2. The evicted edges may disqualify $k - 1$ additional edges touching each of u, w , v, or t from being accounted at C, hence decreasing C by at most $4k - 2$. In total, Φ is decreased by at most $2k - 1$. ■

Lemma 6.4 *A miss of $\mathcal{LRU}$ increases Φ by exactly 1.*

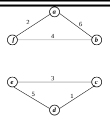
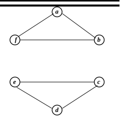
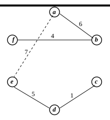
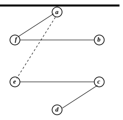
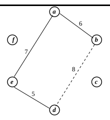
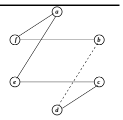
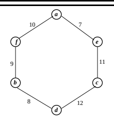
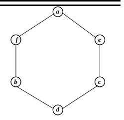
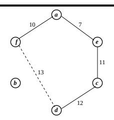
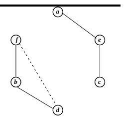
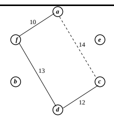
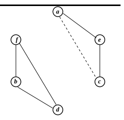
request #	request edge	$\mathcal{LRU}$ m/h	$\mathcal{LRU}$ evict	$\mathcal{LRU}$ state	$\mathcal{OPT}$ m/h	$\mathcal{OPT}$ evict	$\mathcal{OPT}$ state
0							
1	(a, e)	m	(a, f) (e, c)		m	(a, b) (e, d)	
2	(b, d)	m	(c, d) (b, f)		m		
3	(b, f)	m	(a, b)		h		
4	(a, f)	m			h		
5	(c, e)	m	(d, e)		h		
6	(c, d)	m			h		
7	(d, f)	m	(b, d) (b, f)		m	(c, d) (a, f)	
8	(a, c)	m	(a, e) (c, e)		m		
9	(b, f)	m	(a, f)		h		
10	(c, e)	m	(c, d)		h		
11	(b, d)	m			h		
12	(a, e)	m			h		

Table 1: Set of 12 requests for $k = 2$ on which $\mathcal{LRU}$ incurs 12 misses while $\mathcal{OPT}$ incurs only 4 misses. The end configuration is isomorphic to the starting configuration.

request #	request edge	$\mathcal{LRU}$ m/h	$\mathcal{LRU}$ evicted	$\mathcal{OPT}$ m/h	$\mathcal{OPT}$ evicted
1	(a, e)	m	$R_1(a)$, $R_1(e)$	m	(a, b) (e, d)
2	(b, d)	m	$R_1(b)$, $R_1(d)$	m	
3–6	$R_1(a), R_1(b), R_1(d), R_1(e)$	4m	$R_2(a), R_2(b), R_2(d), R_2(e)$	4h	
7–10	$R_2(a), R_2(b), R_2(d), R_2(e)$	4m	$R_3(a), R_3(b), R_3(d), R_3(e)$	4h	
$\cdots$	$\cdots$				
$(4k-9)$-$(4k-6)$	$R_{k-2}(a), R_{k-2}(b), R_{k-2}(d), R_{k-2}(e)$	4m	(a, f), (e, c), (c, d), (b, f)	4h	
$(4k-5)$-$(4k-2)$	same as 3-6 of Table 1				

Table 2: First half of a sequence of requests on which $\mathcal{LRU}$ incurs $2k-1$ times the number of misses incurred by $\mathcal{OPT}$. The sequence generalizes the sequence given in Table 1.

Proof. The requested edge $e = (u, v)$ is cached by $\mathcal{LRU}$, is already cached by $\mathcal{OPT}$, and is the newest edge, hence C is increased by 1. We show that otherwise C and H are unchanged. H (number of holes in $\mathcal{OPT}$'s cache) can not change. Evicted edges can not disqualify other edges from being accounted in C since they can only decrease the set of edges cached by $\mathcal{LRU}$ and not by $\mathcal{OPT}$. To complete the proof, we claim that evicted edges themselves could not have been accounted for in C. Assume the contrary, and suppose that the evicted (u, w) was accounted for in C. Since (u, w) is the "oldest" edge touching u in $\mathcal{LRU}$'s cache, by definition of C all other edges cached by $\mathcal{LRU}$ must be cached by $\mathcal{OPT}$. Since u had no holes, it means that the caches of $\mathcal{LRU}$ and $\mathcal{OPT}$ at u are identical at the "start of the second stage." In particular, the requested edge e is cached by $\mathcal{LRU}$. This contradicts our assumption of a miss. ■

7 Examples

Lemma 7.1 *The number of misses incurred by $\mathcal{LFD}$ and by $\mathcal{LFD}^*$ may be almost twice the optimal.*

Proof. Consider the network shown in Figure 1. The vertices u and v have caches of size k. All other vertices have caches of size 1. Let $A = \langle a_1, \ldots, a_{k-1} \rangle$, $B = \langle b_1, \ldots, b_{k-1} \rangle$, $C = \langle c_1, \ldots, c_{k-1} \rangle$ and $D = \langle d_1, \ldots, d_{k-1} \rangle$. Consider the sequence of requests $ABCDABCD \ldots$, i.e., the sequence $ABCD$ repeated over the over again. $\mathcal{LFD}$ and $\mathcal{LFD}^*$ would both behave in the following way. Initially, the connections $a_1, \ldots, a_{k-1}$ will be placed in the cache of u. When the requests $b_1, \ldots, b_{k-1}$ come, they will be served one at a time using the remaining location in u's cache. Vertex u prefers keeping the connections $a, \ldots, a_{k-1}$ in its cache as they will all be requested again before any one of the b_i's. Unfortunately, when the requests $c_1, \ldots, c_{k-1}$ come, the connections $a_1, \ldots, a_{k-1}$ are dumped from u's cache, as the other endpoints of the $a_1, \ldots, a_{k-1}$ can only have one open connection at a time. Exactly the same happens now with v and the connections $c_1, \ldots, c_{k-1}$ and $d_1, \ldots, d_{k-1}$. As a result, $\mathcal{LFD}$ and $\mathcal{LFD}^*$ suffer a miss for each request. It is possible, however to suffer misses only for about half of the requests. The vertices u and v should simply keep in their caches the connections $b_1, \ldots, b_{k-1}$ and $d_1, \ldots, d_{k-1}$ which will never be dumped, and serve the requests $a_1, \ldots, a_{k-1}$ and $c_1, \ldots, c_{k-1}$ using the remaining slot in their cache. ■

Lemma 7.2 *The performances of $\mathcal{LFD}$ and $\mathcal{LFD}^*$ are incomparable.*

Proof. Consider the network depicted in Figure 2. The center node u has cache size k and all other nodes have cache size 1. Let A be the sequence of requests $\langle a_1, a_2, \ldots, a_{k-1} \rangle$. The sequences B, C, D, and E are defined similarly. It is not difficult to see that on

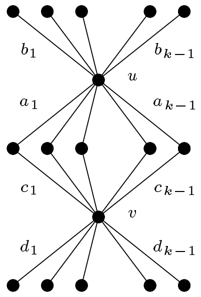


Figure 1: Comparing $\mathcal{LFD}$ and $\mathcal{LFD}^*$ with $\mathcal{OPT}$.

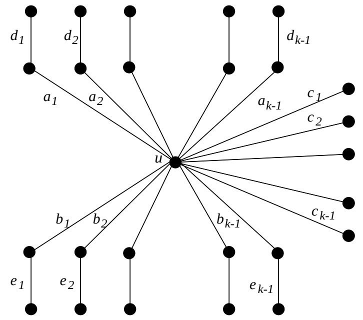


Figure 2: Comparing $\mathcal{LFD}$ with $\mathcal{LFD}^*$.

the repeated sequence ADB, the policy $\mathcal{LFD}$ is optimal with hit rate approaching 1/3 and it outperforms $\mathcal{LFD}^*$, which misses on all requests. On the sequence obtained from repeating $ADEBACEBC$, $\mathcal{LFD}^*$ has hit rate approaching 1/9 and it outperforms $\mathcal{LFD}$, which misses on all requests. ■

Acknowledgements

We thank Balachander Krishnamurthy and Jennifer Rexford for many pointers and Balachander Krishnamurthy, Jeffrey Oldham, and Jennifer Rexford for participating in early discussions on caching connections.

References

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. In *Proceedings of the 4th European Symposium on Algorithms*, pages 419–430. Springer-Verlag, Lecture Notes in Computer Science Vol. 1136, 1996.

[2] Apache HTTP server project. `http://www.apache.org`.

[3] S. Arora and C. Lund. Hardness of approximation. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 399–446. PWS Publishing Company, 1997.

[4] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM systems journal*, 5:78–101, 1966.

[5] A. Borodin and R. El Yaniv. *Online computation and competitive analysis.* Cambridge University Press, 1998.

[6] E. Cohen and H. Kaplan. LP-based analysis of greedy-dual-size. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1999. full version in submission.

[7] E. Cohen, H. Kaplan, and J. D. Oldham. Policies for managing TCP connections under persistent HTTP. In *Proceedings of the World Wide Web-8 Conference*, 1999. to appear.

[8] A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[9] A. Fiat, R. Karp, M. Luby, L. McGoech, D. D. Sleator, and N. Young. On competitive algorithms for paging problems. *J. Alg.*, 12:685–699, 1991.

[10] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proc. 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997.

[11] S. Irani and A. R. Karlin. Online computation. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems.* PWS publishing company, Boston, MA, 1995.

[12] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM'88 Conference*, August 1988.

[13] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *J. on selected areas in communication*, 13, 1995.

[14] S. Keshav and H. Saran. An empirical evaluation of virtual circuit holding times policies in IP-over-ATM networks. In *Proceedings of the IEEE INFOCOM'94 Conference*, 1994. `http://www.cs.cornell.edu/skeshav/papers/hold.ps`.

[15] C. Lund, N. Reingold, and S. Phillips. IP over connection oriented networks and distributional paging. In *Proc. 35th IEEE Annual Symposium on Foundations of Computer Science*. IEEE, 1994. `http://www.cs.cornell.edu/skeshav/doc/94/2-16.ps`.

[16] L. McGoech and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[17] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, August 1997.

[18] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1/2):25–35, December 1995.

[19] C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[20] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list updates and paging rules. *Comm. ACM*, 28:202–208, 1985.

[21] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

[22] N. Young. On line file caching. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms.* ACM-SIAM, 1998.