# Managing TCP Connections under Persistent HTTP


Edith Cohen    Haim Kaplan    Jeffrey D. Oldham*

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

Computer Science Dept.
Stanford University
Stanford, CA 94305
*oldham@cs.stanford.edu*


March 1, 1999


**Abstract**

Hyper Text Transfer Protocol (HTTP) traffic dominates Internet traffic. The exchange of HTTP messages is implemented using the connection-oriented TCP.

HTTP/1.0 establishes a new TCP connection for each HTTP request, resulting in many consecutive short-lived TCP connections. The emerging HTTP/1.1 reduces latencies and overhead from closing and re-establishing connections by supporting *persistent connections* as a default.

A TCP connection which is kept open and reused for the next HTTP request reduces overhead and latency. Open connections, however, consume sockets and memory for socket-buffers. This trade-off establishes a need for connection-management policies.

We propose policies that exploit embedded information in the HTTP request messages, e.g., senders' identities and requested URLs, and compare them to the fixed-timeout policy used in the current implementation of the Apache Web server.

An experimental evaluation of connection management policies at Web servers, conducted using Web server logs, shows that our URL-based policy consistently outperforms other policies, and achieves significant 15%-25% reduction in cost with respect to the fixed-timeout policy. Hence, allowing Web servers and clients to more fully reap the benefits of persistent HTTP.



**Keywords:** HTTP, persistent connections, TCP, connection management, caching


# 1 Introduction

The Hypertext Transfer Protocol (HTTP) dominates information exchange through the Internet. HTTP messages are transported by TCP connections between clients and servers. Most implementations of HTTP/1.0 [6] use a new TCP connection for each HTTP request/response

*Research was performed while visiting AT&T Labs exchange. Hence, the transmission of a page with HTML content and embedded images involves many short-lived TCP connections.

TCP connection is established with a 3-way handshake; and typically several additional round trip times (RTT) are needed for TCP to achieve appropriate transmission speed [34]. Each connection establishment induces user-perceived latency and processing overhead. Thus, *persistent connections* were proposed [31, 19, 32] and are now a default with the draft HTTP/1.1 standard [17, 18]. HTTP/1.1 keeps open and reuses TCP connections to transmit sequences of request/response messages; hence, reducing the number of connection establishments and resulting latency and processing overheads.

Deployment of HTTP/1.1 necessitates policies for deciding when to terminate inactive persistent connections. HTTP/1.1 specifies that connections should remain open until explicitly closed, by either party. Beyond that HTTP/1.1 provides only one example for a policy, suggesting using a timeout value beyond which an inactive connection should be closed [18, §8.1.4]. A connection kept open until the next HTTP request reduces latency and TCP connection establishment overhead. However, an idle open TCP connection consumes a socket and buffer space memory. The minimum size for a socket buffer must exceed the size of the largest TCP packet, i.e., 64Kb, and many implementations pre-allocate buffers when establishing connections [30]. The number of available sockets is also limited. Many BSD-based operating systems have small default or maximum values for the number of simultaneously-open connections (a typical value of 256) but newer systems are shipped with higher maximum values (typically thousands). Studies show, however, that with current implementations, large numbers of (even idle) connections can have a detrimental impact on server's throughput [2, 3, 4].

The design challenge of good connection-management policies is to strike a good balance between benefit and cost of maintaining open connections and to enforce some quality of service and fairness issues. The current version 1.3 of the Apache HTTP Server [1] uses a fixed holding-time for all connections (the default is set to 15 seconds), and a limit on the maximum allowed number of requests per connection (at most 100). (Limiting the number of request was proposed in [19] and provides some fairness.) The Apache implementation is a quick answer to the emerging need for connection management. The wide applicability and potential benefit of good connection-management makes it deserving further study.

Persistent connection management is performed at the HTTP-application layer. Current implementations of Web servers use a *holding-time model* rather than a typical *caching model*. Using holding-times, a server sets a holding time for a connection when it is established or when a request arrives. While the holding-time lasts, the connection is available for transporting and servicing incoming HTTP requests. The server resets the holding-time when a new request arrives and closes the connections when the holding-time expires. In a caching model there is a fixed limit on the number of simultaneously-open connections. Connections remains open ("cached") until terminated by client or evicted to accommodate a new connection request. A holding-time policy is more efficient to deploy due to architectural constraints whereas a cache-replacement policy more naturally adapts to varying server load. Policies in the two models are closely related when server load is predictable [10]; A holding-time policy assigning the same value to all current connections is analogous to the cache-replacement policy LRU (evict the connection that was Least Recently Used). In fact, under reasonable assumptions the holding-time value can be adjusted through time as to emulate LRU under a fixed cache size (and hence adapt to varying server load) [10]. Heuristics to adjust the holding-time parameter were recently proposed and evaluated on server logs [15].

A critical component in the effectiveness of connection-management policies in both models is the ability to distinguish connections that are more likely to be active sooner. LRU exploits the strong presence of reference locality but does not use further information to distinguish between connections. We propose policies based on various attributes of HTTP request messages, and compare them to LRU.

Our policies design was guided by a basic framework, which assumes an associated distribution function on the next-request-time for each open connection. Holding-time values or (analogously) eviction decisions are determined based on these distributions. This framework was previously used in various contexts (for example, holding-times for IP over ATM virtual circuits [25], spinning on a lock in a shared memory multiprocessor [24], adaptive disk spindown [27], and Web proxy caching [12].) Theoretical analysis and polices were given in [10, 24, 27, 28]; an optimal policy in the holding-time model was given in [24, 25, 10]; and an optimal policy in the caching model, for predictable load conditions, was provided in [10]. The optimal policy in this context uses knowledge of the distributions in the best possible way.

Since such distributions are not explicitly available, the framework is deployed using heuristics. Our general methodology is to first identify an attribute of HTTP requests that is correlated with the time to the next-request; For each reasonably frequent value of the attribute, we estimate following inter-request time distribution from a sample data; The estimated distributions are then substituted in the "optimal" policy to assign holding-times. Performance of these derived policies highly depends on the selected attribute, and on the size and relevance of the sample.

An important issue was to obtain a good estimate of the distribution for less-frequent attribute values. We developed a heuristic that uses a weighted-sum of samples from the conditional distribution (for particular attribute value) with a richer sample obtained over all attribute values. This heuristic was critical for the performance of our policies and we suspect it may be valuable in other applications.

Connection management policies can assign timeout values based on attributes present in HTTP requests and responses. For example, requested URL, the referrer URL, and content type. Using the framework above, we obtain and compare policies utilizing different attributes. Our study centered on policies at Web servers. HTTP/1.1 permits either party to terminate a connection. Web servers, however, are typically the busier party, with larger incentives for tighter connection management. Our evaluation was performed using server logs obtained from AT&T's Easy Word Wide Web service [14]. Since HTTP/1.1 is not widely deployed, available logs are for HTTP/1.0 traffic, which do not utilize persistent connections. Deployment of persistent HTTP and pipelining may mostly impact ordering and decrease inter-request times between requests for embedded contents, but longer inter-request times are unlikely to be significantly affected. Thus, changes due to HTTP/1.1 deployment are unlikely to significantly affect our conclusions. To further confront the issue, we simulated the various policies on both the original logs and *clicks logs*, which factor out intervals due to embedded contents, but preserve user think times. Clicks logs attempt to capture traffic patterns with persistent high bandwidth connections.

Our experimental results were consistent across server logs and for both variants of each log; hence, substantiating the soundness and applicability of our conclusions. Our URL-based policy uniformly provided better performance trade-offs than other policies we considered. On the original logs, compared with LRU (or fixed holding-time policy implemented at the Apache server), our adaptive policy used 15%-25% fewer open connections (on average) while incurring the same number of connection establishments; and required 10%-25% fewer connection-establishments when using the same average amount of open connections. Even more significant reductions, typically around 50%, are achieved on the clicks logs. The resulting performance improvement is considerable, since connection-establishments induce user-perceived latency and overhead [16, 31, 23] wheras large number of open connections is both detrimental to throughput [4] and is more likely to reach the server's hard-set limits, causing it to refuse new connections.

In Section 2 we discuss the interaction between HTTP and TCP at Web servers, and the nature of server-logs data. We then describe the logs used in our evaluation. Last, we discuss issues with the use of HTTP/1.0 traffic logs, and our solutions. Section 3 contains our cost model. Section 4 describes how a policy should use an estimate of the distribution of the time to the next request from the same client in making holding-time decisions. This section allows for detailed understanding of our connection management policies. The overall presentation however, permits the reader to proceed with later sections before reading Section 4. Section 5 lists the various connection management policies we evaluated. In Section 6 we provide experimental results and conclusions. Section 7 concludes with proposals for future research directions.

# 2 Experiment Setup and Data

## 2.1 Interaction of HTTP and TCP at Web Servers

The interaction between the operating system and the Web server application imposes some separation between the HTTP session and the transport layer. The HTTP server application writes the response contents into the TCP *send socket.* The actual transmission is performed by the transport layer and in invisible to the application; there is typically no channel for "upward" communication from the transport layer to the HTTP server as to whether and when transmission is completed and acknowledged. Simply put, the HTTP session would hear next from that TCP connection only if and when the connection transports a new HTTP request.

Top-level management of persistent connections is performed at the HTTP server application, and hence, may depend only on information available to the application. In particular, holding-times directives are determined when response contents are placed in the socket, and not, for example, when transmission ends.

Management of persistent connections at the server application amounts to determining a holding-time during which the connection remains open awaiting additional HTTP requests. The server resets the holding-time when a new request arrives.

| label | site characterization | requests | clients | resources |
|---|---|---|---|---|
| L1 | Organization | 946953 | 33101 | 1906 |
| L2 | Multimedia equipment retailer | 1583776 | 60747 | 244 |
| L3 | Fashion retailer | 3921767 | 58838 | 347 |
| L4 | Magazine | 1550162 | 140920 | 245 |

Table 1: Specifications of the server logs.

## 2.2 Server Logs

Our evaluation was performed using Web server logs. For each HTTP request, the logs provide the IP address of the requesting client, the requested URL, a time stamp (in whole seconds), the HTTP response code, the referrer field (if available), and some additional information. The logged time is usually one of the followings: the time when the server starts processing the request, the time when the server starts writing the response contents into the socket buffer, or the time when the server completes writing the response contents into the socket[1]. The times of the other two events are not recorded.

The actual choice which of the three times above is recorded has little significance for our purposes. The elapsed time of the internal server operation of processing the requests and writing the response, is fairly small with respect to transmission time, reasonable holding-time values, and to the 1-second granularity of the time stamps. Therefore, whatever the exact logged time is, it must be close to the time when the HTTP session makes connection management decisions. Further discussion on properties of server logs can be found in [26].

## 2.3 The Data

Our evaluation utilized four server logs obtained from AT&T's Easy World Wide Web [14]. The logs specifications are given in Table 1. Each log contains the stream of requests to the corresponding server from November 21 1997 to February 22 1998. We will refer to these logs using the labels: L1, L2, L3, and L4.

Figure 1(a) shows the cumulative fraction $r(i)$ of the requests for the $i$ most frequently requested resources (URLs) in L3. Figure 1(b) shows the cumulative fraction $c(i)$ of the requests sent by the $i$ clients with the largest number of requests in L3. The corresponding distributions for L1, L2, and L4 are similar. We can see that about 1/5 of the resources are responsible for 80% of the requests where 1/5 of the clients is responsible only for 50% of the requests. Server logs exhibit asymmetry between clients and resources (observed also when the number of resources is large). Most of the requests are made to frequently-requested resources (resource popularity is Zipf like) whereas a large part of requests are made by transient clients. These observations reflect known phenomena and are relevant for designing connection management policies based on client-history and URL.

[1]This varies by server software

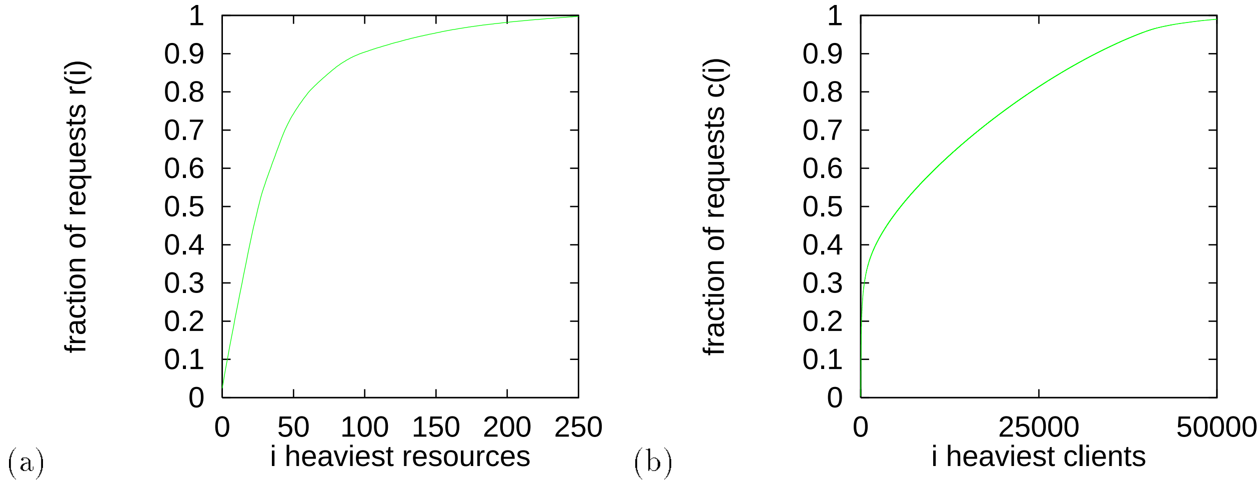



Figure 1: a) Fraction of requests to the $i$ most popular resources (URLs) in L3. b) Fraction of requests by the $i$ most frequent clients in L3.

## 2.4 Using HTTP/1.0 Logs

Available server logs predominantly include HTTP/1.0 traffic, where typically a separate TCP connection is used for each request. Hence, spacings between requests reflect delays due to connection establishment and slow-start [34].

Deployment of HTTP/1.1 is expected to decrease inter-request spacings. Pipelining of requests on the same persistent connection may further decrease spacings. Pipelining is supported by HTTP/1.1 but does not seem to be implemented yet by popular browsers [36]. The decrease in spacings is likely to be more noticeable for shorter inter-request time intervals.[2] Shorter intervals are typically due to requests for embedded contents, whereas longer intervals reflect "user think times" or transmission time. Reasonable connection management policies are likely to maintain persistent connections open for at least a short amount of time following the last request. The TCP TIME_WAIT state [35] implies that closed connections consume resources from 60-240 seconds past their closing time; factoring this in the overhead cost of a new connection establishment, we obtain that the latter compares with open-cost of the order of a minute or more. Thus, supporting the use of longer holding times.

The performance difference between various policies therefore stems from their actions on longer intervals. Hence, the direct use of existing HTTP/1.0 logs is expected to yield meaningful results.

To further substantiate this, we simulate the different persistent connection management policies on both

1. the original HTTP/1.0 logs, and
2. derived *clicks log*.

[2] In the uncommon event (for our data) of significant wait at the server queue, longer intervals are also affected.

The *clicks logs* were derived from the original logs by including only requests due to users actions (requests due to embedded contents were factored out). We developed an effective heuristic that identifies these URLs based on content-type, occurrence in referrer field, and typical length of time since previous request. The click logs approximate a situation where requests for embedded contents always arrive on the same persistent connection, and almost at the same time (due to higher bandwidth and pipelining), as the request for their referrer URL.

We expect future HTTP/1.1 request patterns to be in between the HTTP/1.0 log and the clicks log. Consistent results across both these extremes suggest robustness and applicability of our conclusions.

### 2.4.1 Concurrent TCP Connections

HTTP transfers are often facilitated by a client opening multiple concurrent TCP connections. The maximum number of connections to one server is a browser-configurable parameter, and defaults to 4 concurrent threads [29] in HTTP/1.0 implementations of the Netscape Navigator browser. The suggested number is 2 concurrent persistent TCP connections with HTTP/1.1 [17, 18]. Microsoft Explorer seems to indeed use up to 2 connections whereas Navigator seems to use up to 6 concurrent connections [36].

Available server logs, unfortunately, do not provide information on the number of active connections. The information could be retrieved from the times each HTTP response is received and acknowledged by the client, but these times are not available. Our evaluation therefore assumed a single active connection for each client. We note that our attribute-based policies essentially "learn" policies from sample data, and their performance could only improve if the different connections are distinguished.

# 3 Cost Model

We use the following *holding-time* cost model. Upon receiving an HTTP request $r$, the server decides on a holding-time interval $T(r)$. The server then leaves the connection open for at most $T(r)$ seconds from the moment it received $r$. If a new request $r'$ arrives within the next $T(r)$ seconds, then a new holding-time interval $T(r')$ is in effect. Otherwise the connection is terminated after $T(r)$ seconds.

A *connection management policy* is an algorithm that determines an interval $T(r)$ for every request $r$. Consider a request sequence $s$. The *profit (number of hits)*, $p_A$, of a policy $A$ on $s$ is the number of requests that did not require opening a new connection. The *number of misses*, $m_A$, of $A$ on $s$ is the number of requests that required opening a new connection. The *open-cost*, $H_A$, of a policy $A$ is total time connections were open. The policy used by HTTP/1.0 closes the connection after every request thereby incurring a miss for each request and minimal open-cost. Another extreme is a policy that never closes connections incurring a minimal number of misses and very high open-cost. We are looking for policies between these two extremes, that generate good trade-offs between the number of misses and the open-cost.

We parameterize policies by a threshold $V$ that governs trade-offs between open-cost and number of misses. For all policies considered here, both open-cost and profit increase with $V$, whereas the number of misses decrease with $V$. This follows from the fact that for all policies $T(r)$ increases with $V$ for every $r$. We compare the misses to open-cost trade-offs generated by various policies. The open-cost (when divided by elapsed time) measures the average number of open connections.

Minimizing misses is important since HTTP requests necessitating connection establishments endure longer user-perceived latency; and connection establishments and closings incur processing and memory overhead. The per-connection overhead includes socket buffers allocation/deallocation and memory, periodic processing, and port number consumed when the connection is in TIME_WAIT state, which lasts 60 seconds (for BSD-based implementations) to 240 seconds (as specified in [8]) past the closing time of the connection [35]. On the other hand, small open-cost is important since the number of open connections is subjected to hard-set limits and large numbers of (even idle) open connections degrade server throughput [4].

Keshav et al [25] studied a closely related cost model. A parameter $\mathcal{O}$ governs the relative costs of opening a connection and connection open time. The *cost* of a policy $A$ is defined as its open time plus the total cost for opening connections. That is $C_A = H_A + \mathcal{O} * m_A$. Keshav et al compared the cost of various policies while varying the value of the parameter $\mathcal{O}$. This model is related to ours by capturing the same trade-offs; as we shall see, the policy MPG described in section 4 minimizes $C_A$ when we set the parameter $V$ to be $1/\mathcal{O}$.

Another natural cost model for connection management is a regular *caching model* where we have a fixed capacity $k$ on the number of open connections we can hold. Once a connection needs to be established and the cache is full the policy has to decide and close one of the currently open connections. The goal is to maximize profit.

The holding-time model and the caching model are closely related. There exists correspondence between policies in the two models and their relative performance [10]. If the holding-time policy is adjusted to maintain a near-steady number of open connections, then the open-time cost measure corresponds to cache size (when divided by elapsed time). In [10] it is shown that under reasonable assumptions, experimental results on the relative performance of policies in one model indicate a similar relation between the analogous policies in the other model.

Because of architectural and implementation reasons (see Section 2.1), the first model seems to be a likely implementation for Web server based persistent connection management.

# 4 Using Distributions of Inter-Request Times

In this section we show how a policy can utilize an estimate of the distribution of the time to the next request from the same client. In Section 4.1 we describe a policy *Minimum Profit Gradient* (MPG) to deduce from a inter-request time distribution a holding-time for the corresponding connection. Policy MPG is optimal in the sense that it generates best possible trade-offs between the open-cost and the number of misses among all policies knowing the exact inter-request times distributions. In section 4.2 we show how to use MPG when the exact distributions of the inter-request times are not available and have to be estimated. Our attribute-based policies described later in Section 5.3 are derived using the framework we provide here. These policies differ from each other in the way they associate a distribution with each request, but once distributions are associated, they use used in the same way.

## 4.1 Foundations

Let $D$ be a probability distribution over all possible request sequences. Distribution $D$ defines a set of possible requests where a request $r$ is a triple consisting of a requesting client $c_r$, request time $t_r$, and a distribution function $F_r$. Function $F_r$ is the distribution function of the time to the next request from $c_r$. In this section we assume that the server knows $F_r$ for every request $r$, and describe an optimal policy which a server should use under this assumption.

It is possible to summarize the actions of a policy $B$ with a parameter $V$ by a mapping from the set of all possible requests $S$ (triples $(c_r, t_r, F_r)$) to a probability density function $\Delta_{r,V}(t)$ that captures the likelihood that upon encountering a request at time $t_r$ with associated probability $F_r$, $B$ keeps an open connection to $c_r$ for time period $t$.

Let $r$ be a request from client $c_r$ with associated distribution function $F_r$. The expected profit (number of hits) of keeping a connection to $c_r$ for time length $t$ following $r$ is $\mathrm{Eprofit}_r(t) \equiv F_r(t)$ and the corresponding expected cost (connection time) is $\mathrm{Ecost}_r(t) \equiv \int_0^t (1 - F(x))dx$[3]. It follows from linearity of expectation that the expected profit of $B$ is

$$E(p_B(V)) = \sum_{r \in S} q_r \int_0^\infty \Delta_{r,V}(t) \mathrm{Eprofit}_r(t) dt$$

and the expected open-cost is

$$E(c_B(V)) = \sum_{r \in S} q_r \int_0^\infty \Delta_r(t) \mathrm{Ecost}_r(t) dt \ .$$

Here $q_r$ is the probability of request $r$. That is, the sum over sequences in which request $r$ is made, of the probability of the sequence.

For a request $r$ we define

$$\mathrm{util}_r(t_1, t_2) \equiv \frac{\mathrm{Eprofit}_r(t_2) - \mathrm{Eprofit}_r(t_1)}{\mathrm{Ecost}_r(t_2) - \mathrm{Ecost}_r(t_1)} \equiv \frac{F_r(t_2) - F_r(t_1)}{\int_{t_1}^{t_2} (1 - F_r(x))dx}$$

as the expected profit gain of increasing the caching period from $t_1$ to $t_2$ divided by the expected cost of this increase.

The policy *Minimum Profit Gradient* (MPG) upon receiving $r$ keeps the connection open for $T_V(r) = \max\{z | \forall y < z \ \ \mathrm{util}_r^{-1}(y, z) \leq V\}$. That is, $T_V(r)$ is the maximum point such that for every $y < T_V(r)$, the expected cost of increasing the caching period from $y$ to $T_V(r)$ divided by the expected profit gain from this increase is no greater than $V$.

By the above definition, the policy MPG with parameter $V$ minimizes the expected cost $C_A = H_A + \frac{1}{V} m_A$ when $A$ ranges over the set of all policies that know the probability distribution $F_r$ for every request $r$. This corresponds to choosing $\mathcal{O} = 1/V$ in the Keshav et al cost model mentioned in Section 3.

[3] We shall omit the subscript $r$ when it would be clear from context.

Given a distribution function $F(t)$, there is a discrete set of time intervals that corresponds to choices that MPG can make (on the range of $V \geq 0$ threshold values) for caching a connection to client $c_r$ with associated distribution $F_r(t) \equiv F(t)$. We partition the domain of $F(t)$ into intervals $I_1 = [0, t_1]$, $I_2 = (t_1, t_2]$, $I_3 = (t_2, t_3], \ldots$ where $t_1$ is the maximum $z > 0$ that maximizes the quantity

$$\text{util}(0, z) = \frac{F(z)}{\int_0^z (1 - F(x))dx} ,$$

and $t_i$ $(i > 1)$ is the maximum $z > t_{i-1}$ that maximizes the quantity

$$\text{util}(t_{i-1}, z) = \frac{F(z) - F(t_{i-1})}{\int_{t_{i-1}}^z (1 - F(x))dx} .$$

We define $\text{util}(I_i) \equiv \text{util}(t_{i-1}, t_i)$. It follows from this definition that $\text{util}(I_i) < \text{util}(I_{i-1})$ for every $i$. The following lemma shows that MPG always keeps a connection to $c_r$ for one of the time periods $0 \equiv t_0, t_1, \ldots$.

**Lemma 4.1 ([10])** *An equivalent alternative definition for the choice of policy* MPG *is to keep a connection to $c_r$ for an interval $[0, t_i]$ for the maximum $i$ such that* $\text{util}^{-1}(I_i) \leq V$.

Policy MPG yields optimal trade-offs of profit to open-cost in the sense of the following theorem.

**Theorem 4.1 ([10])** *For every caching policy $B$ that knows the distribution $F_r$ for every request $r$ (but not the actual sequence drawn) there exists values $0 \leq V_1 \leq V_2$ and $0 \leq \alpha \leq 1$ such that $E(H_B) = (1-\alpha)E(H_{\text{MPG}}(V_1)) + \alpha E(H_{\text{MPG}}(V_2))$ and $E(p_B) \leq (1-\alpha)E(p_{\text{MPG}}(V_1)) + \alpha E(p_{\text{MPG}}(V_2))$.*

## 4.2 Deployment

In reality the distributions $F_r$ for each request $r$, are not directly available. We use the following methodology:

- We identify attributes $A_1, A_2, \ldots, A_k$ of requests, that are indicative of next request time.
- For each set of values $u_1, u_2, \ldots, u_k$ of the corresponding attributes, we learn, using available sample data, a distribution $F_{u_1,\ldots,u_k}(t)$ which approximates the conditional distribution on next-request-time following requests with attribute values $u_1, u_2, \ldots, u_k$.
- Our derived policy is the policy MPG, where for a request $r$ with attribute values $u_1, u_2, \ldots, u_k$ we take $F_{u_1,\ldots,u_k}(t)$ as $F_r(t)$.

The derived connection management policy is captured by a mapping $M_V(u_1, \ldots, u_k) \to R^+$ from sets of attribute values to the applicable holding-times. It follows from the above that for any fixed set of values $u_1, \ldots, u_k$, the mapping is a non-decreasing step function of the threshold $V$, where the steps correspond to the interval partition of the distribution $F_{u_1,\ldots,u_k}(t)$.

Performance depends on choice of attributes, in particular, their number, interdependence, and correlation with the next request time. Examples of attributes available with HTTP requests are the requesting client, requested URL, and referrer URL. Our evaluated policies utilizes at most two attributes at a time.

The "learned" distribution is obtained by first collecting a histogram on inter-request times, and then *smoothing* it by taking a weighted average with the general inter-request time histogram.

A good choice of attributes and sample size is such that for most requests, there is a sufficient amount of sample data with the applicable attribute values. With more sample data containing requests for a URL $u$ we are able to obtain more confident estimates for the inter-request time distribution following requests for $u$. For some attribute-values, however, there inherently may not be sufficient data. For example, when considering URLs in fast changing sites or when using the client attribute: Large fraction of requests originate by transient clients which were not seen recently and for which there are only short or no histories available.

A small sample may still carry important information. The following example demonstrates, however, that when small samples are used without smoothing, the effectiveness of distribution-based policies suffers. Suppose we have 4 requests to a URL $u$ in the sample data, with all 4 inter-request times being 2 seconds and below. If the unsmoothened sample histogram is provided as input to MPG, then MPG always selects holding times of at most 2 seconds, since it associates zero expected profit for an increase to a longer holding time. The sample, however, could be a likely outcome from a "true" distribution with 10% likelihood of inter-request times between 2 and 10 seconds; and if so the MPG policy would incurs a miss-rate of at least 10% on all requests made to the URL $u$, even for large values of $V$.

Thus, we deployed the following *smoothing* heuristic to derive the associated distributions. Consider a resource $u$ for which we have $n$ samples. Let $R_u$ be the distribution function obtained by dividing the histogram of inter-request intervals following requests for $u$, by $n$. Similarly, let $G$ be the distribution function obtained in this manner across all available requests. Instead of using the distribution $F_u \equiv R_u$ we used

$$F_u \equiv \frac{n}{n+1} R_u + \frac{1}{n+1} G \ .$$

That is, we use a weighted combination of $R_u$ and the general distribution $G$, where $G$ is weighted as a single additional data point. Note that if we use $G$ as the associated distribution, different requests are not distinguished and the resulting policy collapses to LRU. On the other hand using $R_u$ alone is subject to the problems illustrated in the example above.

Our smoothing scheme is inspired by empirical Bayesian inference methods where the generic distribution $G$ is to be thought of as the prior distribution and $F_u$ is the posterior distribution which we obtain using the prior and the data represented by $R_u$ [7, 20]. Smoothing was crucial for obtaining good policies in our evaluation.

In Section 5.3 we describe the various connection management policies we derived using this framework.

# 5 Policies

## 5.1 LRU

Least Recently Used (LRU) is a common cache replacement algorithm. When a request for a new item arrives and the cache is full, LRU evicts the cached item that was accessed least recently. LRU exploits the high locality of reference usually exhibited in request sequences, and performs well in practice. Classical theoretical results establish that LRU is $\frac{k}{k-h+1}$-competitive against the optimal offline algorithm with cache of size $h \leq k$ [33].

The policy analogous to LRU in the holding-time model is the one that uses the same interval $T(r)$ for every request $r$ [10, 25, 12]. Hereafter we use the name LRU for this policy. LRU is parameterized by the constant $V$ to which $T(r)$ is set.

## 5.2 OPT

As a benchmark we consider the performance of an omniscient optimal policy. The optimal policy knows the sequence in advance and achieves the best trade-offs of misses and open-cost. In the caching mode, Belady's algorithm [5], which evicts the page to be used furthest in the future, is known to be optimal. The optimal policy OPT in the holding-time model is the following. Let $I(r)$ be the length of the inter-request interval from $r$ to the following request. Fixing a threshold $V$, the optimal policy OPT, sets $T(r) = I(r)$ if $I(r) \leq V$ and $T(r) = 0$ otherwise. If $H_{\mathrm{OPT}}(V)$ is the open-cost and $p_{\mathrm{OPT}}(V)$ is the profit of OPT with threshold $V$ then it is easy to see that no policy can have profit exceeding $p_{\mathrm{OPT}}(V)$ for a cost no larger than $H_{\mathrm{OPT}}(V)$.

## 5.3 Attribute-Based Policies

Our attribute-based policies are motivated and derived using the framework described in Section 4. An attribute-based policy for attributes $A_1, A_2$ (e.g., requested URL and referrer URL) is captured by a mapping $M_V^{A_1,A_2} : U_1 \times U_2 \rightarrow R^+$ from possible values of $A_1, A_2$ to holding-times. The connection management policy uses a holding-time of $M_V^{A_1,A_2}(u_1, u_2)$ following all requests with attribute values $A_1 = u_1$ ,$A_2 = u_2$. The parameter $V$ corresponds to the relation between the cost of establishing a new TCP connection and the cost of maintaining an idle open connection for a period of time. For any fixed set of values, the holding-time is non-decreasing with $V$ and is actually a step function. Performance trade-offs for each policy are obtained by sweeping $V$. We evaluated policies derived from the following attributes.

- **Requested URL** (RESOURCE).
  The policy is captured by a mapping $M_V^{res}(u)$. A holding-time of $M_V^{res}(u)$ is used after all requests for a URL $u$.

- **Referrer Resource** (REFERRER).
  The mapping $M_V^{ref}(u)$ is defined for each URL $u$ that occurs as a referrer. A holding-time of $M_V^{ref}(u)$ is used after all requests with a referrer URL $u$.

- **Referrer and Requested URL** (RES-REF).
  The policy is characterized by the mapping $M_V^{\{res,ref\}}(u,v)$, which is defined for each pair of URLs $(u,v)$ such that $v$ appears as the referrer URL in some requests for $u$. A holding-time of $M_V^{\{res,ref\}}(u,v)$ is used after all requests made for a URL $u$ with a referrer URL $v$.

- **Resource Size** (SIZE).
  Requests are characterized by the response size. We partition the set of possible sizes into bins. The bin index is the attribute which we use to derive this policy. The mapping $M_V^{size}(j)$ is defined for each bin index $j$. Holding-time of $M_V^{size}(j)$ is used after all requests made for a resource with response size in bin index $j$.

- **Client history** (CLIENT)
  The policy is characterized by a mapping $M_v^{\{client,hist\}}(c,h)$ For each client $c$, and history-length $h$.

  The holding-time used for the $j$th request for a client $c$ is $M_V^{\{client,hist\}}(c,h)$, where $h < j$ is largest such that $M_v^{\{client,hist\}}(c,h)$ is defined.

  The CLIENT policy uses the same information as Jacobson's *estimated exponentially weighted mean and variance* policy [21]. Jacobson's policy, and also a version of our CLIENT policy was evaluated by Keshav et al [25] for circuit holding-times of IP traffic over ATM.

### 5.3.1 Learning and Test Data

The mapping describing each attribute-based policy is constructed using log data. The construction can be performed either online or by a periodic off-line processing of a subset of the server log. An online construction may be timely whereas an off-line one minimizes run-time computation overhead.

Our evaluation was for an off-line construction. We partitioned the logs into two parts of approximately the same size. Each part contained the request sequences of (a random) half of the clients. Policies were evaluated on one part of the log ("test data"). For all but for the CLIENT attribute, the mappings were computed once using the other part of the log ("learning data").

Sensitivity analysis described in Section 6 shows that policies derived from learning data containing only a small fraction of clients yield comparable results. We observed that access patterns in our logs were fairly static through time. Hence, our off-line construction obtained good results (learning data derived on-line from very-recent history is likely to be more effective for more dynamic sites).

For CLIENT we simulated an online policy utilizing the accumulated available history of each client. This online construction is more appropriate for the CLIENT attribute, since clients are very transient, and many requests are due to clients that only had a short interaction with the server over a long period of time (see Section 2.3). Hence, a periodic analysis would not provide sufficient per-client data.

### 5.3.2 Computation Cost

The computational cost of constructing a policy (deriving the mapping $M_V$) is linear in the size of the learning data and involves a single pass. A histogram of inter-request times is collected for every set of attribute values $(u_1, u_2) \in U_1 \times U_2$. A generic histogram, across all requests, is also collected.

The resulting policy is represented by the mapping $M_V^{A_1,A_2}(u_1, u_2)$ from sets of attribute-values to holding-times. The mapping can be stored in a hash-table to facilitate a quick assignment of holding-times to requests. The size of the hash-table depends on the number of distinct sets of values considered, and can widely vary for different attributes and between Web sites.

# 6 Experimental Results and Conclusions

Figures 2 and 3 plot the performance of LRU, OPT, and the attribute-based policies RESOURCE, CLIENT, REFERRER, RES-REF, and SIZE on the original logs. Figures 4 and 5 show the performance of the same policies on the *Clicks Logs*, derived from the original logs by considering only inter-request times due to user actions (see Section 2.4).

The $y$-axis (*miss rate*) represents the fraction of requests for which a persistent connection was not available (In other words, the ratio of connections establishments to requests). The measurements are restricted to requests where the previous request by the same client had occurred no longer than 10 minutes before. The $x$-axis (*average open time per request*) corresponds to the total combined time connections were open divided by the total number of requests. Note that this corresponds to the average number of open connections divided by the average request rate.

As expected, the plain LRU policy was outperformed by attribute-based policies. The RESOURCE policy was consistently the best (or nearly the best) performer, across all server logs, and for both the original logs and the clicks logs. The policies REFERRER and SIZE often approach the performance of RESOURCE, but never exceed it.

The observations that REFERRER performs close to RESOURCE, but that the combined policy RES-REF does not generally outperform RESOURCE, indicates that the REFERRER attribute indeed carries a lot of the information present in the RESOURCE attribute, but does not carry significant *additional* useful information for predicting lengths of inter-request intervals. This is due to the fact that on the logs we considered, most requests for a typical resource (URL) had the same referrer. Hence, by and large, values of the resource attribute constitute a finer sub-partitioning of the partition obtained by values of the referrer attribute.

The SIZE policy partitioned response sizes to about 30 bins each corresponding to a range of sizes. The bins were mapped to holding-time intervals. If response size is indeed correlated with inter-request time, we expect assigned holding-times to generally be longer for bins of larger response sizes. A close look at these mappings, however, shows that this is far from being the case. The actual explanation to the good performance of SIZE is the fact that most requests in a typical bin were due to a small number of URLs (sometimes a single URL), and hence, SIZE was actually mimicking the RESOURCE policy on these bins.

The client-history policy CLIENT yields marginal improvements, if at all, over LRU, indicating that unlike the situation in other contexts (e.g., IP over ATM [25]), each client's overall history with the server is not a significant hint for predicting the clients future inter-request intervals. The identity of the requested URL provided a much more meaningful hint for the duration of the inter-request interval. The CLIENT policy also suffers because typically there are many transient clients each having a single or few short interactions with the server. Thus, there is a very short history available for a large fraction of the requests. On such requests, the policy applies the generic holding-time and CLIENT reduces to LRU.

The consistent results obtained across all servers and for both the clicks logs and the original logs suggest that our conclusions and results are fairly robust. In particular, we expect them to be similar for data that incorporates persistent connections. Results were also consistent across a wide range of open-cost values, suggesting robustness for varying load and server capacities.

The overall-best policy, RESOURCE, achieved significant improvements over LRU. Consider the logs L1, L2, L3, and L4, respectively, and the use of a fixed 15 seconds holding-time interval (the default in the current implementation of the Apache HTTP server code). The corresponding "miss rates" are 12.4%, 17.6%, 13.2% and 13.2%. The respective normalized open-costs are 4.05 seconds, 5.13 seconds, 4.28 seconds, and 5.6 seconds per request. The RESOURCE policy achieves the same miss rate with normalized open-costs of 3.22, 4.38, 3.5, and 4.2 seconds per request, respectively. Hence, achieving reductions of 20%,15%, 16%, and 25% in total open time (equivalently, in average number of open connections) with respect to LRU.

The reduction in cost with respect to LRU is even more significant on the clicks logs. On the L4 log, for example, fixed holding-times of 34 and 51 seconds result in respective normalized costs of 30 and 43 seconds, and miss rates of 51% and 37%, respectively. The RESOURCE policy achieves these miss rates with respective costs of 15 and 23 seconds, which constitutes improvements of about 50% over LRU. As can be seen from Figures 4 and 5, other logs exhibit similar results.

When a relatively small number of resources dominates a large fraction of requests, as is typical in server logs, a smaller amount of learning data suffices for RESOURCE to perform well. Figures 2–5 plot performance of policies when the learning data set included half the clients. Figure 6 shows that performance does not degrade when the learning data contains only 1/32 of the clients. The mapping of resources to respective holding-times is constructed in linear time in the size of the learning data. Therefore, using less data also simplifies the computation of the mapping. Another observation is that the Web sites represented in our logs were fairly static; therefore, a single computed policy was effective for the total period of 3 months. We expect more dynamic sites to benefit from more frequent re-computation of the policies.

# 7 Future Research Directions

The emergence of persistent HTTP and the cost and benefit trade-offs of keeping open idle TCP connections necessitate the deployment of connection-management policies. We believe

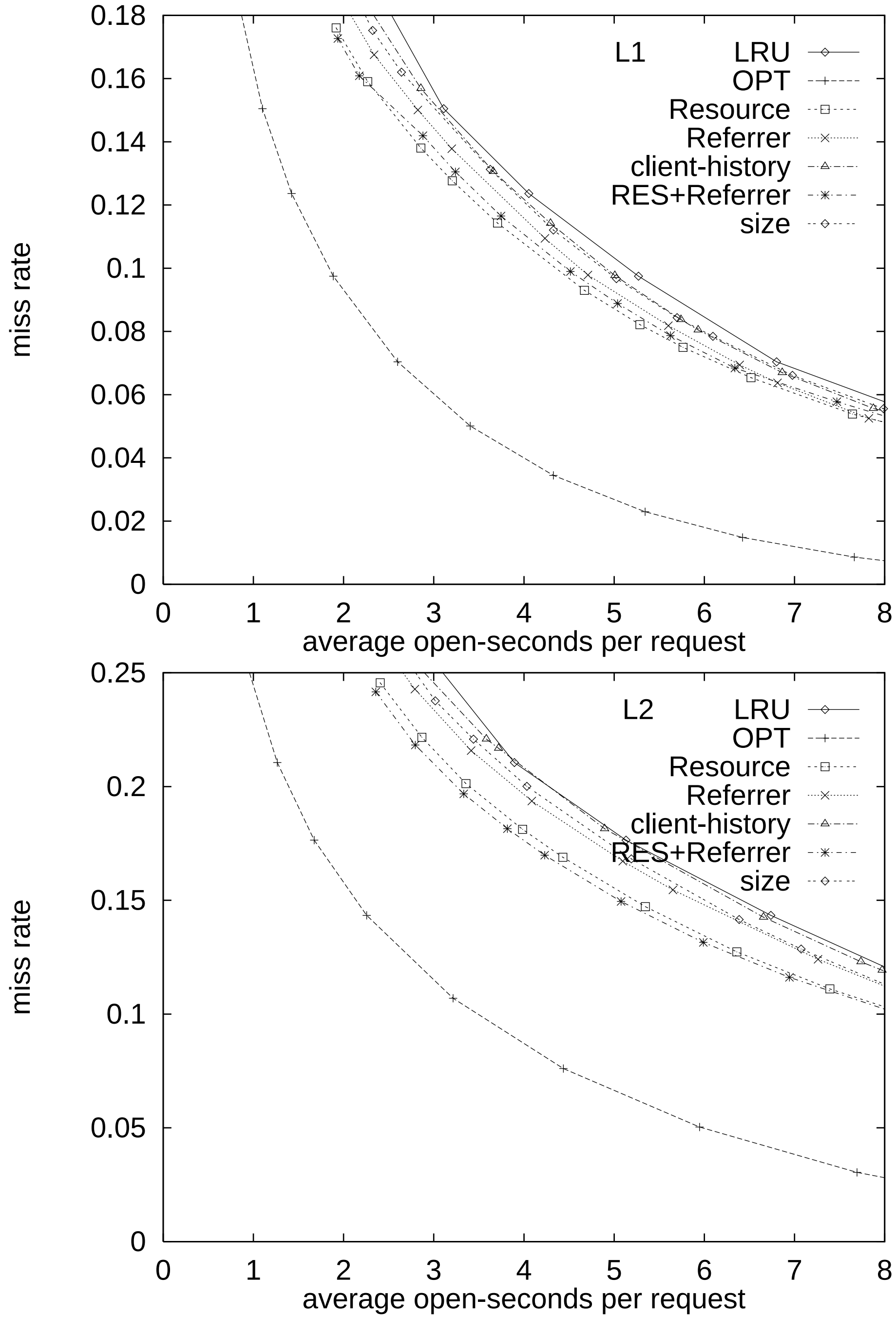



Figure 2: Performance of the various connection management policies on the L1 and L2 logs. The y-axis shows the fraction of new connections established among requests occurring at most 10 minutes following a previous request by client. The x-axis shows the total open time normalized by total number of requests.

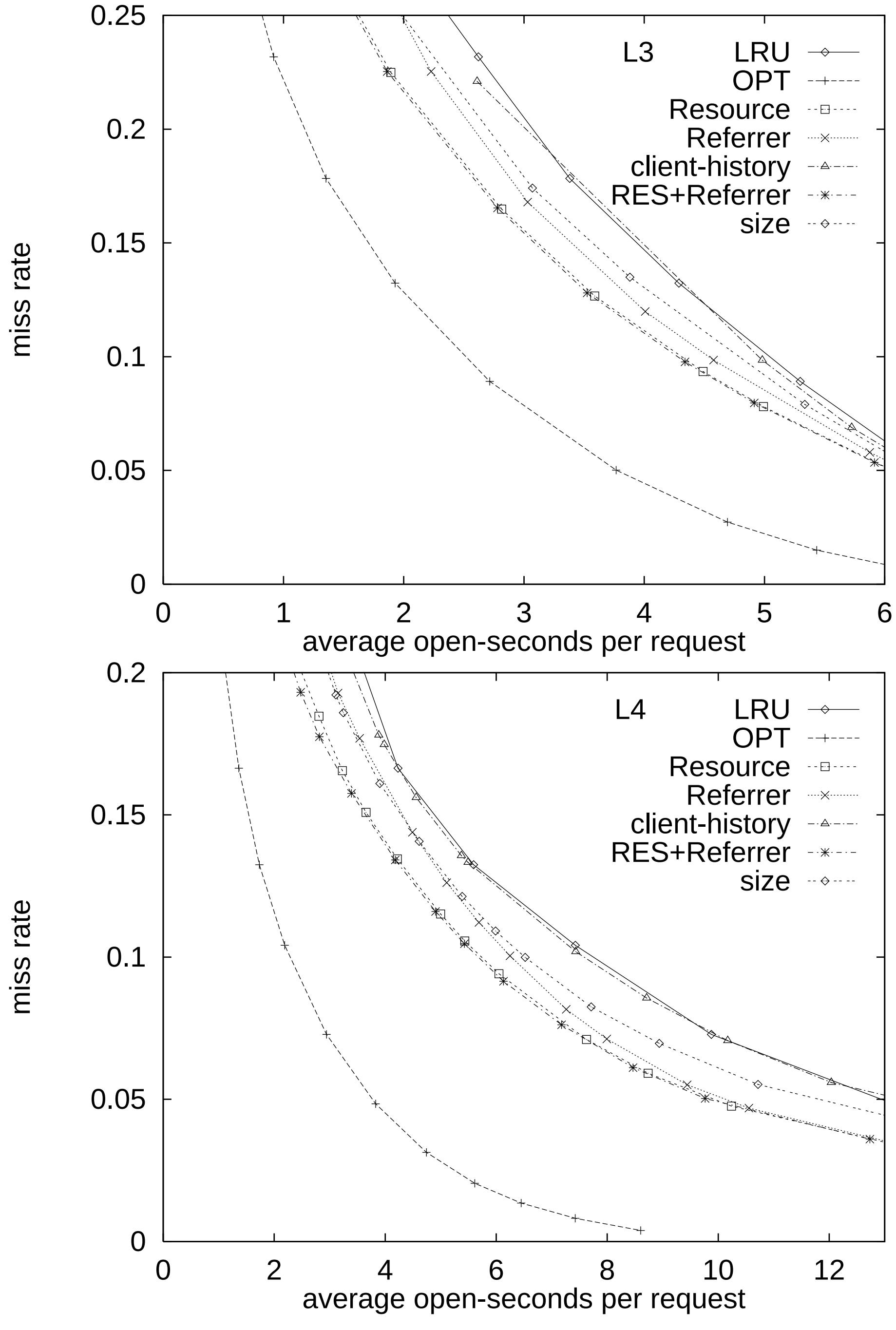
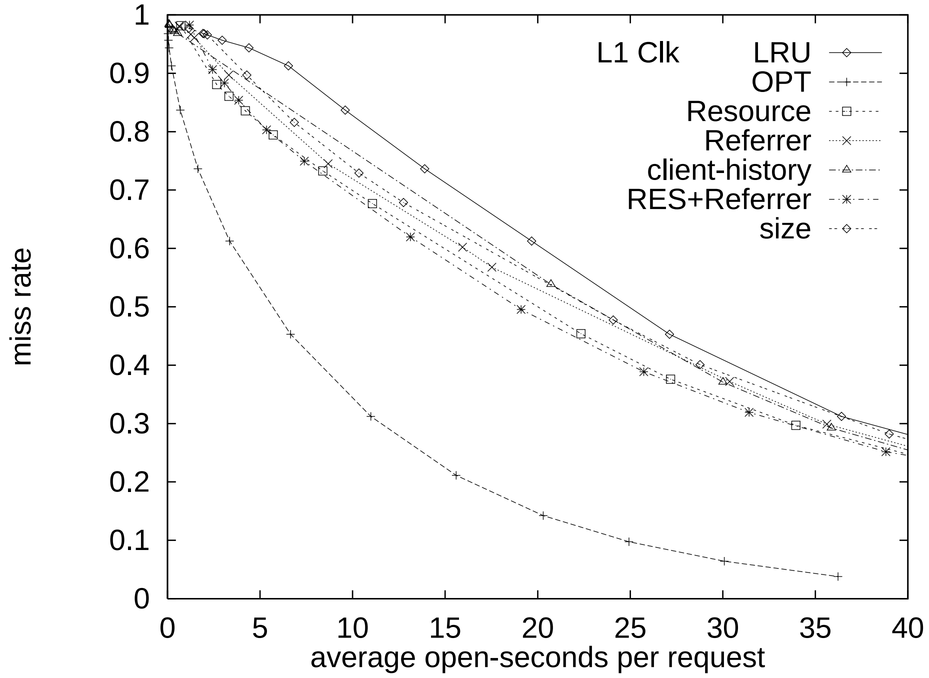



Figure 3: Performance of the various connection management policies on the L3 and L4 logs. The y-axis shows the fraction of new connections established among requests occurring at most 10 minutes following a previous request by client. The x-axis shows the total open time normalized by total number of requests.



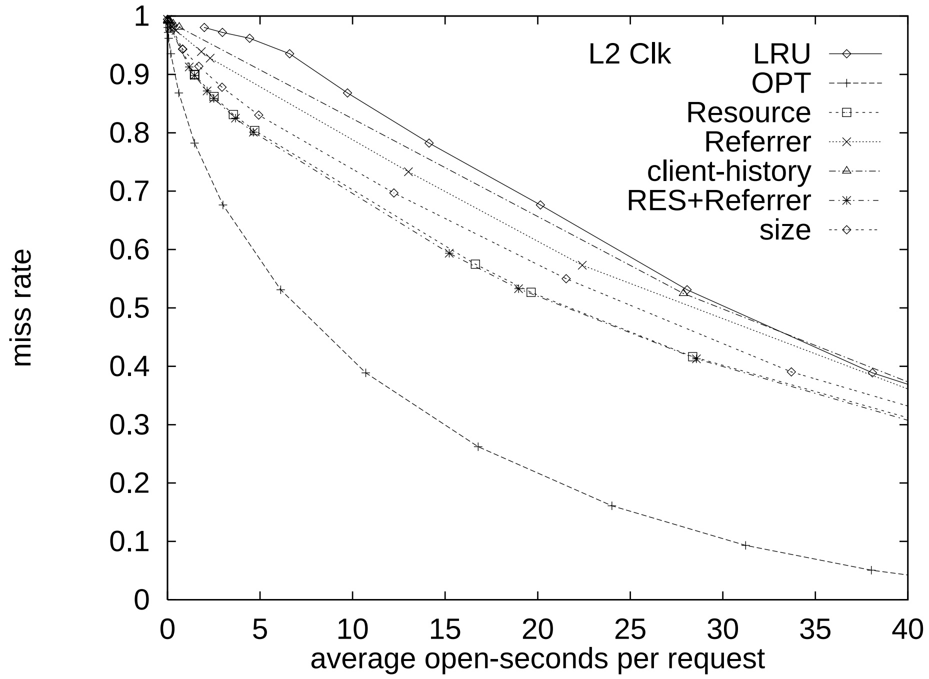




Figure 4: Performance of the various policies on the L1 and L2 clicks logs.

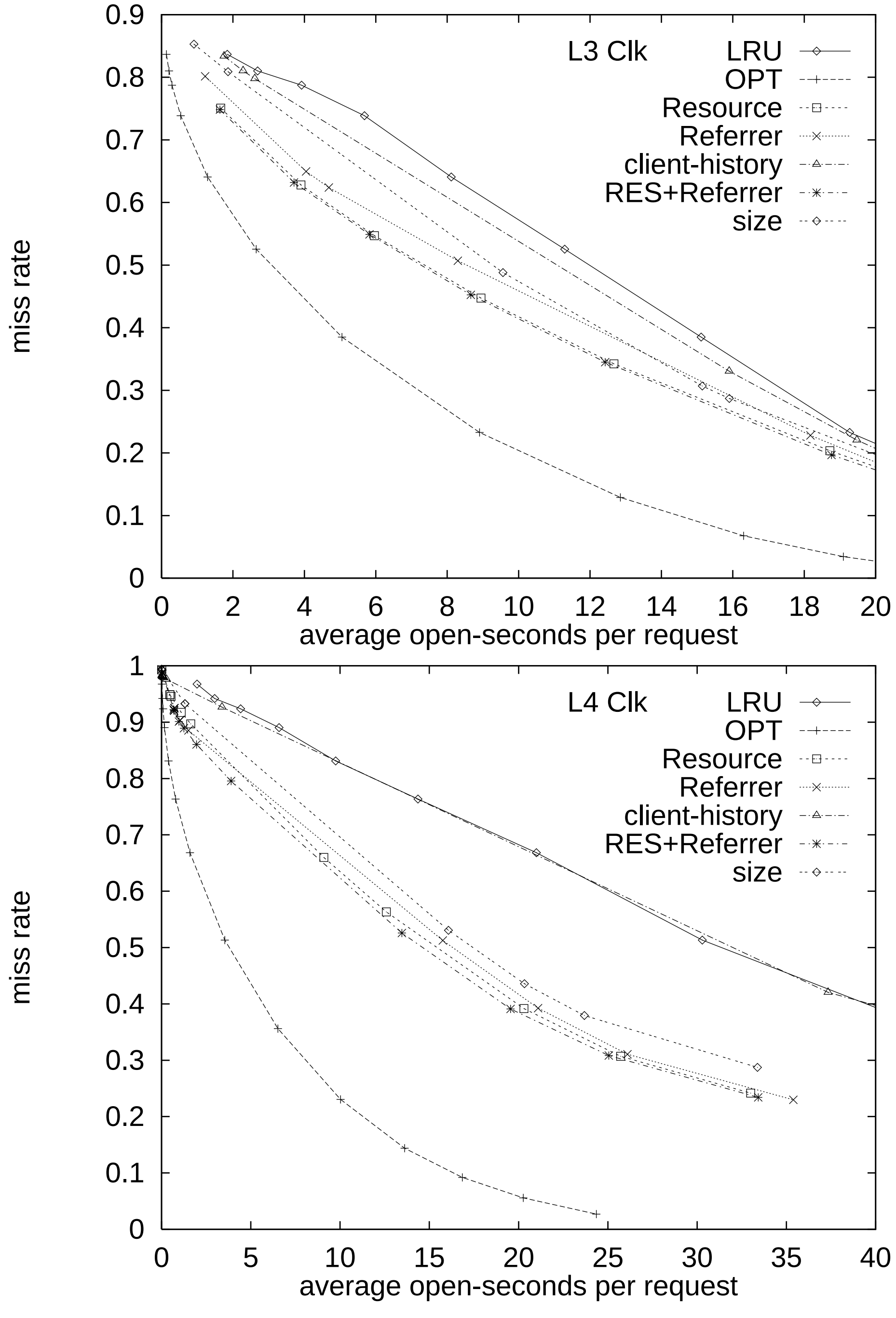



Figure 5: Performance of the various connection management policies on the L3 and L4 clicks logs.

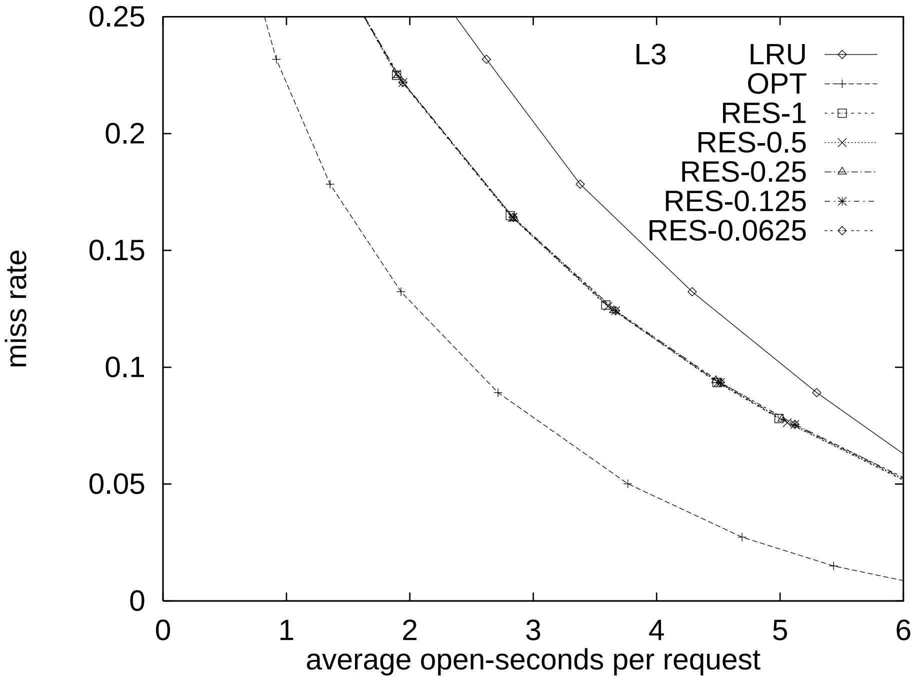



Figure 6: Performance of the RESOURCE policy on the L3 server log, where the learning data included all logged requests made by 1/2, 1/4, 1/8, 1/16, and 1/32 of the clients.

that the study and design of connection-management policies can potentially improve performance to the same extent that paging and cache-replacement policies contributed in the contexts of operating systems and Web.

We proposed and evaluated several adaptive policies for managing persistent connections at HTTP Web Servers. We concluded that a policy which determines holding-times based on the requested URL outperforms other policies and significantly improves over fixed-length holding-times (LRU). Beyond connection management, our methodology of deriving policies is general and may find applications in other domains.

We suggest several extensions for future research. Our evaluation assumed uniform connection-establishment costs. Varying costs, however, can support several quality of service classes or capture perceived-latency that varies with different network RTTs. Under varying establishment costs, the objective is to optimize trade-offs of total open-time and combined connection establishment costs. An analogous extension for document caching is when documents have varying *fetching costs*. In this context, Young [37, 38] proposed a generalization of LRU termed GreedyDual. An experimental study on Web proxy traces was performed in [9]. GreedyDual generalizes LRU and can be adopted for connection management. The MPG policy extends to handle varying establishment costs [10]. Thus, our attribute-based policies can be generalized as well.

Our policies are by and large Markovian, in the sense that holding-time depends on attributes of the preceding requests. Such policies are simple to represent and deploy, and achieved good performance. One obvious question is the gain from extending the viability of holding-times beyond the next request.

Our study focused on connection management at Web servers. HTTP/1.1, however, allows both server and client to unilaterally terminate a connection. And indeed, connection management is deployed at proxy servers and browsers as well. Popular browsers seem to deploy different connection-management policies: Internet Explorer uses a fixed 60 seconds timeout for idle persistent connections whereas Netscape Navigator uses an LRU-managed fixed-size cache of 15 connections [36]. If connections are well-managed at every host, the busier party is likely to initiate termination. Busy proxy servers support TCP connections on behalf of many users; and proposals for HTTP-NG[4] [22] support connection re-use *across* users. A recent study indicates that connection caching and re-use at proxy caches may reduce user-perceived-latencies more than document caching [16][5]. Thus, proxy-side connection management emerges as an important challenge.

Connection management at proxy servers differs in several respects from server-side management: Both request time and the time response is received are available to the HTTP session layer and can be used by the policy. There is more per-user information available, since all of each user's activities, across all servers, are viewed by the proxy, but there is considerably less per-resource or per-server information. Hence, the resource-based policy that was very effective for server-side management is not likely to be effective at the proxy's end, client-based policies, however, may be effective. Another avenue is to bridge the server-client information gap using piggybacking techniques as proposed in [13].

In principle, the two-sided control of each persistent connection could lead to global underutilization. Accommodating a new connection may cause the closing of two existing connections, one at each end. Moreover, a connection closed by one end may be a priority at the other end. Such interactions suggest studying the global behavior of local connection management algorithms. Recently, Cohen et al [11] proposed a theoretical model for connection caching. Using this model they prove bounds on the competitive-ratio of local algorithms such as LRU against the optimal offline (not local) algorithm.

## Acknowledgments

We thank Yoav Freund, David Johnson, Yishay Mansour, Jeff Mogul, Balachander Krishnamurthy, and Jennifer Rexford for helpful discussions and references, and Balachander Krishnamurthy for providing the data. The presentation benefited from valuable suggestions made by the WWW8 referees.

## References

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

[4] "Next Generation" HTTP also referred to as HTTP/2.0.

[5] The study in [16] was based on a fixed holding times of 30 seconds.

[2] G. Banga and P. Druschel. Measuring the capacity of a Web server. In *Proceedings of the USENIX Symposium on Operating on Internet Technologies and Systems*. USENIX Association, 1997. `http://www.cs.rice.edu/~gaurav/papers/web-paper.ps`.

[3] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (ODSI)*. USENIX Association, 1999. `http://www.cs.rice.edu/~gaurav/papers/osdi99.ps`.

[4] G. Banga and J. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1998. `http://www.cs.rice.edu/~gaurav/papers/usenix98.ps`.

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

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, MIT/LCS, May 1996. `http://ds.internic.net/rfc/rfc1945.txt`.

[7] G. E. P. Box and G. C. Tiao. *Bayesian Inference in Statistical analysis*. John Wiley & Sons, New York, 1992.

[8] R. T. Braden. Requirements for internet hosts – communication layers. RFC 1122, ISI, October 1989.

[9] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
`http://www.usenix.org/events/usits97`.

[10] E. Cohen and H. Kaplan. Exploiting regularities in Web traffic patterns for cache replacement. In *Proc. 31st Annual ACM Symposium on Theory of Computing*. ACM, 1999.

[11] E. Cohen, H. Kaplan, and U. Zwick. Connection caching. In *Proc. 31st Annual ACM Symposium on Theory of Computing*. ACM, 1999.

[12] E. Cohen, B. Krishnamurthy, and J. Rexford. Evaluating server-assisted cache replacement in the Web. In *Proceedings of the 6th European Symposium on Algorithms*, pages 307–319. Springer-Verlag, Lecture Notes in Computer Science Vol. 1461, August 1998.

[13] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings of the ACM SIGCOMM'98 Conference*, September 1998.

[14] AT&T Easy World Wide Web.
`http://www.ipservices.att.com/wss/hosting`.

[15] M. Elaoud, C. J. Sreenan, P. Ramanathan, and P. Agrawal. Use of server load to dynamically select connection closing time for HTTP/1.1 servers. Submitted for publication, March 1999.

[16] 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.

[17] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2068, MIT/LCS, May 1997. `ftp://ds.internic.net/rfc/rfc2068.txt`.

[18] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1, November 1998. `ftp://ftp.ietf.org/internet-drafts/draft-ietf-http-v11-spec-rev-06.ps`.

[19] H. Frystyk Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, August 1997.

[20] I. J. Good. *The Estimation of Probabilities: An Essay on Modern, Bayesian Methods*. MIT Press, Cambridge, Massachusetts, 1965.

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

[22] B. Janssen, H. Frystyk, and Spreitzer M. HTTP-NG architectural model, August 1998. `http://info.internet.isi.edu/in-drafts/files/draft-frystyk-httpng-arch-00.txt`.

[23] H. Kaplan and E. Cohen. Reducing user-perceived latency by prefetching connections and pre-warming servers. Submitted, 1999.

[24] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[25] 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.

[26] B. Krishnamurthy and J. Rexford. Software issues in characterizing web server logs. In *W3C Web characterization group workshop*, 1998. position paper.

[27] P. Krishnan, P. M. Long, and J. S. Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. *Algorithmica*, 23:31–56, 1999.

[28] 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`.

[29] A. Luotonen. *Web proxy servers*. Prentice-Hall, Englewood Cliffs, New Jersey, 1998.

[30] J. Mogul, October 1998. personal communication.

[31] J. C. Mogul. The case for persistent-connection HTTP. *Computer Communication Review*, 25(4):299–313, October 1995.
`http://www.research.digital.com/wrl/techreports/abstracts/95.4.html`.

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

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

[34] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, Reading, MA, 1994.

[35] W. Richard Stevens. *TCP/IP Illustrated*, volume 3. Addison-Wesley, Reading, MA, 1994.

[36] Z. Wang and P. Cao. Persistent connection behavior of popular browsers.
`http://www.cs.wisc.edu/~cao/papers/persistent-connection.html`.

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

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