ROPE: The Rutgers Online Proxy Evaluator

Brian D. Davison and Chandrasekar Krishnan
Department of Computer Science
Rutgers, The State University of New Jersey
110 Frelinghuysen Road
Piscataway, NJ 08854-8019 USA
davison,krishnan@cs.rutgers.edu

August 2001

Abstract

The Simultaneous Proxy Evaluation (SPE) architecture provides one way to measure the performance of proxy caches. It includes the novel ability to compare prefetching proxy cache performance, but poses a number of implementation challenges. In this report we describe our prototype implementation of SPE, the Rutgers Online Proxy Evaluator (ROPE). We discuss a number of issues raised during development, describe validation tests, and demonstrate the use of our prototype in two experiments to simultaneously evaluate up to four publicly available proxy cache implementations. We measure bandwidth used and response latencies, but also discover unexpected caching bugs in two of the proxies tested.

1 Introduction

There are many possible methods to evaluate caching algorithms and implemented proxy caches [16]. Of those that examine real systems, the typical approach is to apply benchmark tests (e.g. [8, 5, 4, 9, 29, 23, 31]) to either measure performance with a standard load, or to determine the maximum load attainable. The current *de facto* process is to compare many proxies using the same Polygraph [31] workload (e.g., in a caching competition [32]). Unfortunately, these approaches are unusable for evaluating proxy caches that preload content (such as [13, 12]).

A preloading proxy cache typically chooses what to preload based on either user retrieval history or the content of recently retrieved pages. Neither are modeled well (if at all) in artificial workloads and datasets. In the case where the workload is a captured trace from real users, the content is typically unavailable (at least not in the form that the users saw). Additionally, since a preloading proxy cache may choose to preload content that is never actually used, a captured trace will not be able to provide resource characteristics such as size or retrieval time.

For these reasons, we have proposed an approach using simultaneous proxy evaluation (SPE), potentially using a live network and client workload[18]. In this technical report we provide a progress report on our development effort toward an initial prototype implementation of SPE, which we call ROPE (the Rutgers Online Proxy Evaluator), and our experiences with it. In the next sections we briefly review the SPE architecture, followed by a discussion of related efforts. We then describe our application-layer ROPE implementation, and discuss issues that arose from the implementation and how we resolved them. Since ROPE is a new, untested system, we provide some baseline experiments in Section 7 to help validate its performance. In Section 8, we demonstrate the

use of ROPE by detailing two sample experiments that evaluate multiple proxies simultaneously. We wrap up with a discussion of future work and conclusions reached from this effort.

2 Simultaneous Proxy Evaluation

A variety of proxy caches exist in the market. (Web sites such as [20] provide a detailed list of products and vendors.) The features and performance provided often vary more than the marketing literature would suggest. As more and more Internet users access the Internet through proxy servers everyday, these factors play a significant role in determining the quality of the Internet. Hence, it becomes important to study the relative merits and demerits of these servers. There are many characteristics of proxy caches that can be evaluated:

- The peak and average number of requests that can be serviced by the proxy cache per unit time.
- The object hit ratio — the number of objects that are serviced from the proxy's cache against the total number of objects served by the proxy.
- The byte hit ratio — the sum of the bytes served from the proxy's cache against the total bytes served by the proxy.
- The average latency as experienced by the user.
- Bandwidth used by the proxy. In addition to being affected by hit rates, a prefetching proxy is likely to use more bandwidth than a non-prefetching proxy.
- Robustness of a proxy.
- Consistency of the data served by a proxy.
- Conformance to HTTP standards.

Many of these are implementation characteristics, and thus can only be evaluated by techniques that work with complete systems. The Simultaneous Proxy Evalaution architecture was proposed to work at this level, providing a number of benefits in comparison to existing techniques. They include:

- SPE takes existing proxy caches and uses them as black boxes for evaluation purposes.
- SPE tests proxies on real world data.
- SPE eliminates variation in measured performance due to changes over time in network and server loads.
- SPE allows for the evaluation of content-based prefetching proxies.

Thus the SPE architecture is concerned with many of the characteristics of proxy cache performance: bandwidth usage, retrieval latencies, robustness and consistency. In the following sections we summarize the SPE architecture and discuss its implementation in the Rutgers Online Proxy Evaluator (ROPE) prototype.

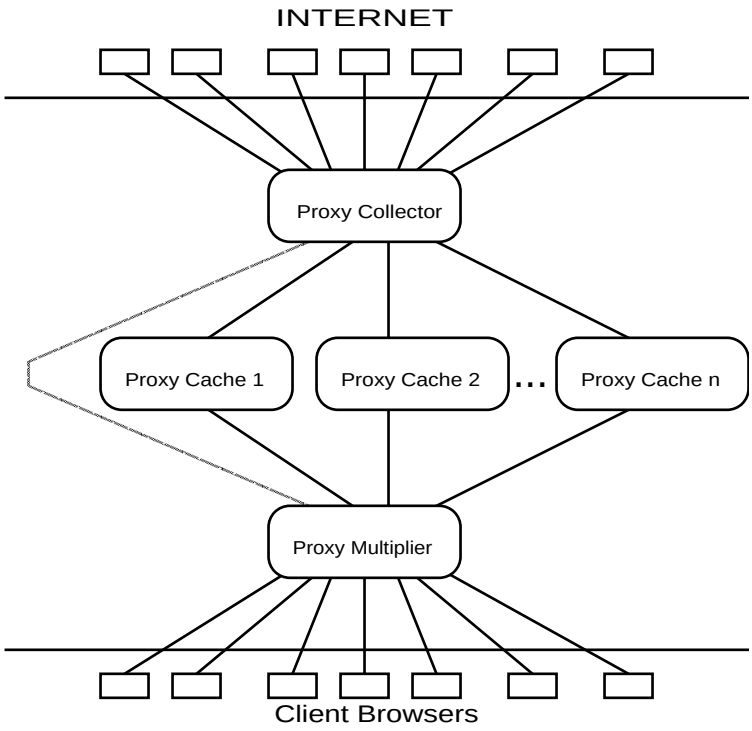

Figure 1: The SPE architecture. All client requests are sent to the proxy Multiplier, which sends a copy to each proxy cache being evaluated. The proxy Collector is a specially configured cache (able to temporarily store both cacheable and uncacheable responses).

3 The SPE Architecture

Figure 1 depicts the SPE architecture. Clients are configured to connect to a non-caching proxy, the *Multiplier*. Requests received by the Multiplier are sent to each of the proxies which are being evaluated, and directly to the Collector to test for stale proxy responses. Instead of letting the test proxies retrieve the objects directly from the origin server, which increases the traffic in the network, they are configured with a parent proxy cache, the *Collector*, which interacts with the origin server. This prevents two types of problems that might otherwise arise: 1) an increase in traffic loads on the network or at the origin server, and 2) side-effects from multiple copies of non-idempotent requests [21, 19]. The Multiplier also sends the requests directly to the Collector to use the responses to service the client requests and validate the responses from the test proxies.

Thus the Multiplier does not perform any caching or content transformation. It forwards copies of each request to the Collector and the test proxies, receives responses, performs validation and logs some of the characteristics of the test proxies. The clients view the Multiplier as a standard HTTP proxy.

Only minimal changes are required in clients, the proxies, or the origin servers involved in the evaluation. The clients need to be configured to connect to the Multiplier instead of the origin server and with popular browsers this is easy to do. The test proxies need to be configured with the Collector as the parent proxy. No changes are required with the Web servers which ultimately service the requests.

In contrast, the Collector is a proxy cache with additional functionality:

- *It will fake the time cost of a miss for each hit.* By doing so we make it look like each proxy has to pay the same time cost of retrieving an object.
- *It must cache normally uncacheable responses.* In this way we prevent multiple copies of uncacheable responses from being generated when each tested proxy makes the same request. Each is cached only for a small amount of time (the *freshness time*), since they are not meant to be cached at all and we cache them in the expectation that they will be requested by other test proxies shortly.

Each proxy tested thus sees a single proxy (the Multiplier) as client, and a single proxy (the Collector) as parent. If the tested proxies only cache passively, then the overall bandwidth used by the system should be exactly that which is needed to serve the client requests. If one or more proxies perform active revalidation or prefetching, then the overall bandwidth used will reflective the additional traffic.

The architecture attempts to tolerate problems with the test proxies. Problems like non-adherence to protocol standards, improper management of connections, and the serving of stale responses can be caught and logged by the Multiplier. Note that while the SPE architecture is designed to preserve safety for passive proxy caches (allowing just one copy of a request to go to the origin server), it cannot provide guarantees of safety for non-demand requests that use the standard HTTP/1.1 GET mechanism [19].

The use of SPE also has some drawbacks, including:

- The Multiplier and Collector will introduce additional latencies experienced by users. However, our expectation is that this will be less than that of a two-level cache hierarchy (since the Multiplier does not do any caching).
- Tests using SPE may not be exactly repeatable, but this is true of any evaluation using a live network or request load.
- The Multiplier and Collector must be able to manage at least $n+1$ times as many connections as any individual tested proxy, where n is the number of simultaneous proxies being tested.

For these reasons the SPE architecture is not proposed for use in testing for peak workload performance or to determine proxy failure modes. Thus, the capabilities complement, but do not supercede, existing proxy performance evaluation techniques, such as those we describe below.

4 Related Work

A number of researchers have proposed proxy cache (or more generally just Web) benchmark architectures (e.g., the Wisconsin Proxy Benchmark [5, 4], WebMonitor [2, 3], hbench:Web [28], and Surge [10]). Some use artificial traces; some base their workloads on data from real-world traces. They are not, however, principally designed to use a live workload or live network connection, and are generally incapable of handling prefetching proxies.

Web Polygraph [31] is an open-source benchmarking tool for performance measurement of caching proxies and other content networking equipment. It includes high-performance HTTP clients and servers to generate artificial Web workloads with realistic characteristics. Web Polygraph has been used to benchmark proxy cache performances in Web cache-offs [32].

Koletsou and Voelker [26] built the Medusa Proxy, which is designed to measure user-perceived Web performance. It operates similarly to our Multiplier, in that it duplicates requests to different Web delivery systems and compares results. It also can transform requests, e.g., from Akamaized URLs to URLs to the customer's origin server. The primary use of this system was to capture the usage of a single user and to evaluate (separately) the impact of using either: 1) the NLANR proxy cache hierarchy, or 2) the Akamai content delivery network. The first test is quite similar in concept to that made by Davison [17] as a partial application of the SPE architecture, which also explored the potential for use of the NLANR hierarchy to reduce retrieval latencies, but which assumed the use a small local proxy cache in between the browser and the hierarchy. While acknowledging the limitations of a small single-user study, their paper also uses a static, sequential ordering of requests – first to the origin server, and then to the NLANR cache. The effects of such an ordering (such as warming the origin server) are not measured. Other limitations of the study include support only for non-persistent HTTP/1.0 requests and fixed request inter-arrival time of 500ms when replaying logs.

Liston and Zegura [27] also report on a personal proxy to measure client-perceived performance. Based on the Internet Junkbuster Proxy [25], it measures response times and groups most requests for embedded resources with the outside page. Limitations include support for only non-persistent HTTP/1.0 requests, and a random request load.

Liu *et al.* describe experiments measuring connect time and elapsed time for a number of workloads by replaying traces using Webjamma [24]. The Webjamma tool plays back HTTP accesses read from a log file using the GET method. It maintains a configurable number of parallel requests, and keeps them busy continuously. While Webjamma is capable of sending the same request to multiple servers so that the server behaviors can be compared, it is designed to push the servers as hard as possible. It does not compare results for differences in content.

While all of the work cited above is concerned with performance, and may indeed be focused on user perceived latencies (as we are), there are some significant differences. For example, our approach is designed carefully to minimize the possibility of unpleasant side effects — we explicitly attempt to prevent multiple copies of a request instance to be issued to the general Internet (unlike Koletsou and Voelker). Similarly, our approach minimizes any additional bandwidth resource usage (since only one response is needed). Finally, while the SPE Multiplier can certainly be used for measuring client-side latencies if placed adjacent to clients, it has had a slightly different target: the comparative performance evaluation of proxy caches.

5 ROPE Implementation

Accompanied by a few simple log analysis tools, the Multiplier and Collector are the two major pieces of software that make up ROPE. In our implementation the Multiplier is a stand-alone multi-process program which is compatible with HTTP/1.1 [21] and HTTP/1.0 [11] protocols and was developed from scratch. The publicly available Squid proxy server [34] version 2.3 was modified to develop the Collector.

5.1 The Multiplier

The Multiplier listens on a configurable port for TCP connections from clients and forks a child process for every client connection opened.[1] Connections to the tested proxies are handled by this

[1] Actually, we first built a multi-threaded version but had trouble using it in our testing environment which had a Linux 2.4 kernel added to RedHat 7.0. When calling `pthread_exit`, the child thread would end up killing the parent.

Sample HTTP/1.0 request:

```
GET http://news.bbc.co.uk/fn.gif HTTP/1.0
Referer:  http://news.bbc.co.uk/
Proxy-Connection:  Keep-Alive
User-Agent:  Mozilla/4.76 [en] (WinNT; U)
Host:  news.bbc.co.uk
Accept: image/gif, image/x-xbitmap, image/jpeg
```

Sample HTTP/1.0 response:

```
HTTP/1.0 200 OK
Date:  Mon, 26 Mar 2001 05:42:00 GMT
Server:  Apache/1.3.12 (Unix)
Last-Modified:  Fri, 10 Dec 1999 15:35:39 GMT
ETag:  "2c137-65f-38511dcb"
Content-Length:  1631
Content-Type:  image/gif
Proxy-Connection:  keep-alive
```

Figure 2: Sample HTTP/1.0 request from Netscpe and response via a Squid proxy.

process using `select(2)`. When a complete request is read, the process opens a TCP connection with the Collector and test proxies, sends the request to them (the HTTP version of this request is the same as the one sent by the client) and waits for more requests (in the case of persistent connections) and responses. Persistent connections are allowed if the client requests it (the version of Squid used to build the Collector supports persistent connections). However, the test proxies are not required to support persistent connections. If they do not, a new connection is opened every time a client sends a request over a persistent connection in the same session. An alternative approach is to not support persistent connections at all, which is clearly not desirable when some proxies do support persistent connections. In any case, the cost of establishing a connection will be recorded each time for a proxy which does not support persistent connections.

Figure 2 shows the headers of a sample request and response. The Multiplier parses the status line of the request and response to record the request method, response code and HTTP version. The only other headers that are manipulated during interaction with client and proxies are the connection header ("Connection" and the non-standard "Proxy-Connection") to determine when the connection is to be closed and the content length header ("Content-Length") to determine when a request or a response is complete. The Multiplier adds a "Via" header to the requests it sends in conformance with HTTP/1.1 standards.

The Multiplier measures various features like the time to establish connection, the size of object retrieved, the initial latency, the total time to transfer the entire object and the MD5 checksum [30] of the object. A data structure with this information is maintained and once all the connections are closed, they are logged, along with the results of validation of responses. Validation of responses are performed by comparing the status codes, the header fields and the MD5 checksums of the responses sent by collector and each test proxy. Mismatches, if any, are recorded. Figure 3 shows some sample lines from the Multiplier log.

5.2 The Collector

Modifications proposed in Section 3 were made to the publicly available Squid proxy cache to develop the Collector. The *StoreEntry* is the Squid data structure which stores information about every object that is cached by the Squid cache. We added four new fields to the StoreEntry structure, which record the cost of retrieval of the object: *Initial Response Time (IRT)*, *Average*

```
Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Collector:  0:34195 0:34090 5083 {DUMMY,} 192.168.0.2
non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-1:  0:108105 0:50553 5083 {200,200,OK,}
192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-2:  1:180157 0:3868 5083 {200,200,OK,}
192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-3:  0:113010 0:34160 5083 {200,200,OK,}
192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/
```

Figure 3: Sample Multiplier log showing the timestamp, the initial latency and transfer time in seconds and microseconds, the size of the object in bytes, the result of response validation, the client IP, the HTTP version, the method, and the URL requested. The possible results are OK (if validation succeeded), STATUS_CODE_MISMATCH (if status codes differ), HEADER_MISMATCH (if headers differ, along with the particular header) and BODY_MISMATCH (if the response body differ).

Byte size (ABS), *Average Chunk Time (ACT)* and *Number of Chunks (n)*. If a request arrives for an object which is not cached, let t_{req} be the time when the request was sent to the origin server, t_0 be the time when the response started arriving. Let the response be received in n chunks of sizes s_1, s_2 ... s_n at times t_1, t_2 .... t_n. Then Initial Response Time is defined as $t_0 - t_{req}$. Average Byte Size is defined as $\sum_{i=1}^{n} s_i/n$. Average Chunk Time is defined as $\sum_{i=1}^{n}(t_i - t_{i-1})/n$.

t_{req} is recorded in Squid's *fwdConnectStart* function in *forward.c*, which is called when a connection to origin server is started. t_0 is recorded in the *fwdConnectDone* function in *forward.c*, which is called once a connection with an origin server is established successfully. Each s_i and t_i are recorded in the *storeAppend* function in *store.c*, which is called whenever a chunk of data is added to a StoreEntry. The ABS and ACT are computed from these values once the retrieval of the object is complete, in *httpReadReply* function in *http.c*.

When a request is received for an object which is stored in the cache, the cached copy is returned to the client (e.g., a tested proxy) after waiting for IRT, in n chunks of ABS bytes each, with an interval of ACT between them.[2] The function *clientSendMoreData* in *client_side.c* takes care of sending a chunk of data to the client. We have modified it so that for cached objects, the appropriate delays are introduced.

To cache uncacheable objects in Squid, two modifications are required. The algorithm which determines the cacheability of responses has to be modified to cache these special objects. In addition, when a subsequent request is received for such an object, the cached object should be used instead of retrieving it from the origin server, since these objects would be normally treated as uncacheable. Squid's *httpMakePrivate* function in *http.c* marks an object as uncacheable and the object is released as soon as it is transferred to the client. Instead of marking it private, we modified the code so that the object is valid in the cache for a configurable amount of time (the freshness time). Subsequent requests for this (otherwise uncacheable) object would be serviced from the cache if they are received before the object expires in the cache. The Collector also keeps track of the clients which have seen a particular uncacheable object with a bit mask. If a client has already retrieved this object and makes a request for it again before its expiration, the object will be fetched from the origin server (since such a client would want a fresh copy).

[2]Actually, Squid has a maximum transmit buffer of 4KB, so in the case that ABS is larger, we use 4KB instead, and adjust the delays accordingly so that the total response time is preserved.

```
986185568.949 677 128.6.60.20 TCP_MISS/200 13661 GET http://news.bbc.co.uk/cele0.jpg -
DIRECT/news.bbc.co.uk image/jpeg

986185568.965 2 128.6.60.79 TCP_HIT/200 13660 GET http://news.bbc.co.uk/cele0.jpg - NONE/-
image/jpeg

986185569.404 1914 128.6.60.20 TCP_MISS/200 34543 GET http://news.bbc.co.uk/1254610.stm -
DIRECT/news.bbc.co.uk text/html

986185570.593 926 128.6.60.79 TCP_HIT/200 34550 GET http://news.bbc.co.uk/1254610.stm - NONE/-
text/html
```

Figure 4: Sample Collector (Squid) log showing the timestamp, elapsed time in ms, client IP address, cache operation/status code, object size, request method, URL, user ident (always '-'), upstream source, and filetype.

If an object is validated with the origin server when an If-Modified-Since (IMS) request is received and a subsequent non-IMS request is received for the same object, then to fake a miss, the cost for the entire retrieval of the object is used and not the cost for the IMS response. While handling POST requests, an MD5 checksum of the body of the request is computed and stored when it is received for the first time. Thus in subsequent POSTs for the same URL, the entire request has to be read before it could be determined whether the cached response can be used to respond to the request. Similarly, requests with different cookies are sent to the origin server, even though the URLs are the same. Figure 4 shows some lines from a sample Collector log (identical to a standard Squid log).

6 Issues in implementation

In this section, we make some general remarks about the SPE architecture and discuss a number of the issues encountered during implementation and how they were addressed:

- **Additional client latency.** From Figure 1, it is clear that the object retrieved must traverse at least two more connections than if it is retrieved by the client directly from the origin server —- one between the client and the Multiplier and the other between the Multiplier and the Collector. Though this introduces additional latency on the client's side, we expect it to be small since the ROPE setup typically runs in the same local network as the client. We will evaluate the overhead introduced by the ROPE by running the ROPE within another ROPE in section 7.3.

- **Request scheduling.** If the Multiplier forwards requests to the Collector and the test proxies immediately after receiving a request from a client, there is a good chance that a test proxy will request an object from the Collector, whose transfer from the origin server has not started at all or whose response headers are still being read. In both these cases, the Collector has to open a parallel connection with the origin server to service the second request, since it does not have information about the cacheability of the object and assumes it is safer to retrieve it separately. This is clearly not desirable, since we only want only one request to be sent to the origin server for every request sent by the client. Our implementation of the Multiplier addresses this problem by sending the request only to the Collector first and upon receiving the complete response from the Collector, sends the request to the test proxies.

Thus we ensure that when a test proxy requests an object from the Collector, the transfer of the object is complete and hence, duplicate requests are avoided (because the Collector will have cached the response).

- **Connection costs.** Since the SPE architecture is typically realized on a LAN, the connection setup costs for HTTP connections between tested proxies and the Collector will not mirror those between the Collector and the origin server. Instead, our approach is to capture the connection setup time to the origin server, and add it to the delay replicated on a simulated miss.

  The cost of opening a connection to retrieve an object may not be directly available, if it is not the first request in a persistent connection. This cost may be needed if one of the test proxies does not support persistent connections and the cost of opening a connection has to be included in calculations while faking a miss for this proxy. Hence, the Collector records the cost of opening a connection in the StoreEntry structure for the first object in the persistent connection. This version does not record this connection time in the data structure of objects subsequently requested on this persistent connection (which would be ideal).

- **Request pipelining.** Pipelining of requests (having more than one outstanding request at a time) is allowed by HTTP/1.1, but it requires some additional complexity in the client side, since if a transfer fails, it needs to keep track of the failed requests and reissue them. In ROPE, the Multiplier sends pipelined requests to the Collector only when the client generates them. Thus, the client is responsible for reissuing the failed requests. Even though we accept them, we do not send pipelined requests to the test proxies and make sure that there is only one outstanding request at any point of time (for a single connection — if a browser opens multiple connections to the Multiplier, each is treated independently).

- **Response timings at application level.** As described above in the Collector implementation, the response rate from the Collector for faked misses is the average response rate of the origin server. This, along with the connection setup costs, differs from real-world conditions. However, our implementation has been strictly at the application level, and better replication of connection costs and data transfers would need TCP driver support.

- **Freshness time.** The freshness time for uncacheable objects is set to a small number since we expect the requests for them from other proxies to be sent soon and we don't want to cache them for a long time — they are not supposed to be cached at all. In some cases, the object could be requested by a test proxy after its expiration in the cache (if one of the proxies prefetched an object, it turned out to be uncacheable and the client actually makes a request for it later). In this case, it is retrieved again from the origin server. We cannot avoid this by increasing the freshness time since the Collector would be more likely to return stale data.

- **Distinct IP addresses.** In this version of ROPE, test proxies must be operated on machines with distinct IP addresses as the Collector uses them to maintain the response mask (of which proxies have seen a particular response).

7 Validation of ROPE

ROPE is a new SPE implementation, and as such, deserves some skepticism until it has been proven to be accurate and reliable. In this section, we describe our system configuration and three

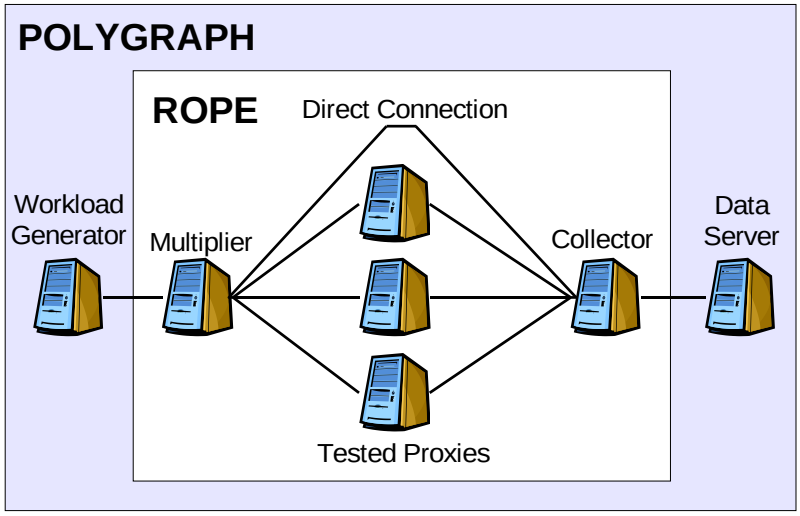

Figure 5: The artificial workload configuration.

experiments that test and provide limited validatation of our implementation.

7.1 Configuration

In the validation tests and in the experiments, we will use up to four proxy cache configurations. All are free, publicly available with source code. They are: Squid 2.3.STABLE4 [34], Apache 1.3.14 (using mod_proxy) [7], WcolD 961127_143211 [13], and DeleGate 7.3.1 [33]. Each used default settings where possible. We selected a uniform 200MB disk cache for each (except for DeleGate which does not appear to support a maximum disk cache size). The source for each was unmodified except when needed to allow compilation on our development platforms and to fix a few WcolD bugs. In addition, for many tests we will also run a "Dummy" proxy, which is just our Multiplier configured to do nothing but pass requests on to a parent proxy.

For each test we run a Multiplier and a Collector as described above, with a freshness time of three minutes. In addition, many tests also employ a load generator and data source, as depicted in Figure 5. For this we employed Polygraph to run five robots, each at one request per second. We used the standard "SimpleContent" Polygraph workload and "olcStatic" for the object life cycle. Object sizes were exponentially distributed with a 13kb mean, and were assigned to be cacheable 80% of the time.

Each proxy (except for the Collector) runs on a separate Dell OptiPlex, as do each of the monitoring and load-generating processes. Each machine has a single 800Mhz Pentium III with 256MB RAM and a single 18GB IDE drive. All machines are connected via 100Mhz ethernet with a shared hub using private IP space. A mostly stock RedHat 7.0 was installed on each machine. The Collector ran on a similar system but with a 933Mhz Pentium III and 512MB RAM, and was dual-homed on the private network and on the Rutgers LAN for external access.

7.2 Proxy penalty

Here we are concerned with the internal architecture. We want to know whether ROPE imposes extra costs for the proxies being tested, as compared to the direct connection. The experiment we chose is to force both the direct connection and tested proxy to go through identical processes. We ran the Polygraph workload for close to twenty minutes (almost 5500 requests) to drive the ROPE

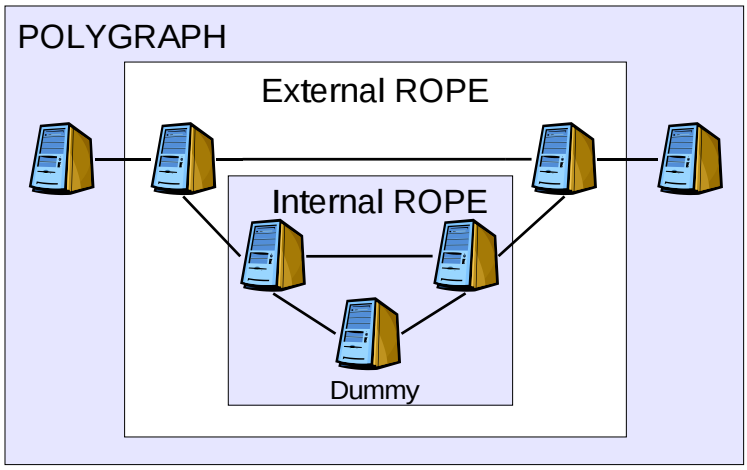

Figure 6: Configuration to test ROPE within ROPE.

architecture, which fed requests through two Dummy proxies. Since both proxies are running identical software on essentially identical machines, we can determine the difference in how the architecture handles the first (normally direct) connection and tested connections.

In this experiment we found that the direct connection was reported to be about 35ms faster on average (mean of 216ms vs. 260ms, and medians of 142ms vs. 180ms). From this result (of an admittedly small test), we might consider subtracting a factor of around 35ms from the measured times of tested proxies.

7.3 ROPE overhead

In this experiment we are concerned with the general overhead that ROPE introduces to the request/response data stream. We want to know how much worse performance will be for the users behind the ROPE implementation, given that each request will have to go through at least two additional processes (the Multiplier and the Collector).

The idea is to test a ROPE implementation using a ROPE implementation, as shown in Figure 6. We used Polygraph to generate the artificial workload for fifteen minutes (just over 4500 requests) to drive the external ROPE. The inner ROPE (being measured) had just one Dummy proxy to drive. The outer ROPE measured just the inner ROPE.

We found that the inner ROPE implementation generated an average response time of 417ms, as compared to the direct connection response time of 185ms. Medians were smaller, at 290ms and 98ms, respectively. Thus, we estimate the overhead (at least in this configuration) to be on average between 192ms and 232ms seconds, before factoring out the 35ms recognized in the previous experiment.

7.4 Proxy ordering effects

As described earlier, the Multiplier in our ROPE implementation waits until it gets a response from the Collector, and then proceeds to make the same request to the tested proxies, in the order as defined by the ROPE configuration. In reality, our implementation is more careful. On a new request that has just received the response from the Collector, the Multiplier will sequentially go through the list of tested proxies. If a persistent connection to that proxy has already been

Proxy name	Relative BW-sent	Relative BW-rec	Responses delivered	% responses match direct	% responses not matching	Mean latency	Median latency
(direct)	100%	100%	19740	100.00%	0.00%	266ms	179ms
Squid	100%	34.4%	19740	99.96%	0.04%	203	97
DeleGate	98.0%	32.8%	19656	85.72%	14.28%	606	209
Dummy	99.6%	99.6%	19602	99.95%	0.05%	429	327
Apache	98.0%	32.4%	19584	85.67%	14.33%	243	117

Table 1: Artificial workload results.

established in this process, the Multiplier uses it and sends the request, and moves on to the next proxy. Otherwise, the Multiplier issues a non-blocking connect to the proxy, and moves on. When the sequence is complete, the Multiplier waits for activity on each connection and responds appropriately (i.e., receiving and checking the next buffer's-worth of data, or sending the request if a new connection has just been established).

In this test we are concerned about the potential effects of the strict ordering of tested proxies. We want to make certain that there is no advantage to any particular position for a proxy being tested. Again we generated the artificial workload for just over ten minutes (approximately 3100 requests) through ROPE to a set of identically configured Apache proxy caches.

In this experiment we found that all three proxies had very similar performance. Each generated data differing from the original in approximately 12% of the responses. This corresponds to a believed caching bug, which we discuss further in Section 8.1.2. Mean response times were 248ms, 242ms, and 244ms, with medians of 121ms, 125ms, and 120ms, respectively. An additional run of this experiment similarly generated results that again differed by only a few milliseconds.

8 Experiments Using ROPE

For the experiments using ROPE to test proxies[3], we employed the same configurations as described in Section 7.1, except where noted.

8.1 Proxies driven by synthetic workload

Here we wish to test the performance of various proxies using a well-known workload generator. For this experiment we tested Squid, DeleGate, Apache, and a Dummy proxy.

8.1.1 Workload details

We used the Polygraph workload as described earlier to generate a simple synthetic workload. This dataset does not contain real (or even parsable) content, and is thus not suitable for a content-based prefetching proxy, and so we are using only passive proxies for this test. We ran the Polygraph workload for a little over an hour (close to 20,000 requests).

8.1.2 Results

The results of this experiment are shown in Table 1 and graphically displayed in Figure 7. The timings shown are unadjusted results from ROPE. Squid comes very close to perfect on comparisons of

[3]In accordance with the terms of the Web Polygraph license under which we are permitted to publish Polygraph-generated results, the raw logs from which we calculate performance in this section can be obtained from the first author, or online (currently from `http://www.cs.rutgers.edu/~davison/pubs/2001/dcs-tr-445/`).

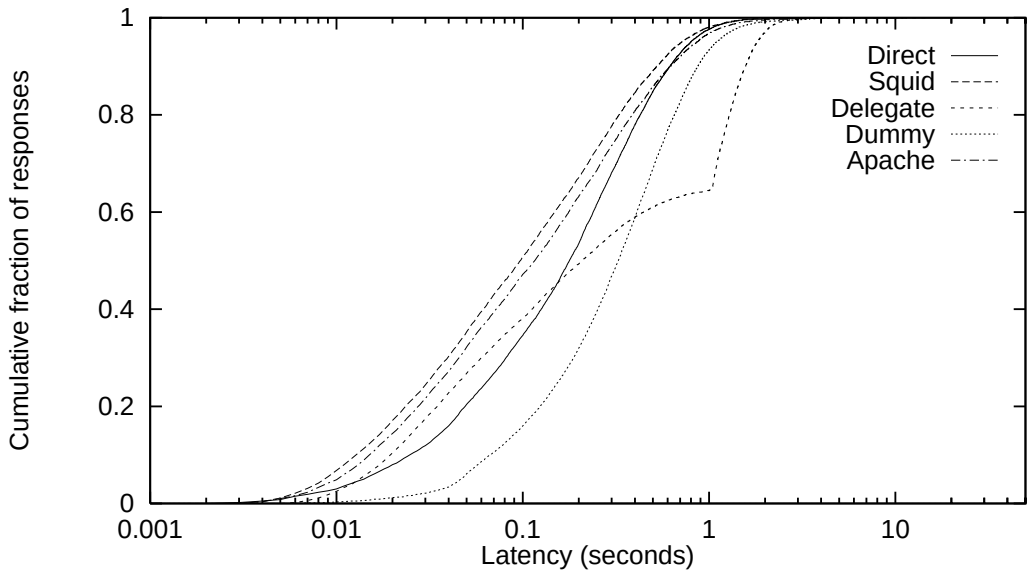

Figure 7: The cumulative distribution of response times for each proxy under the artificial workload.

quality and responds to all requests. The Dummy proxy, however, misses over a hundred responses and generates responses that do not match in a small fraction of cases. DeleGate misses less than ninety responses, while Apache misses over 150. Both Apache and DeleGate generate results in over 14% of responses that do not match the original. This is a significant fraction, and so we looked further. We tested both Apache and DeleGate using the same Polygraph workload directly (that is, without ROPE at all), and found that Polygraph complained similarly of non-cacheable responses being served from the cache.[4]

In terms of response time, Squid is best, by far, with a median response time of just 97ms (203ms mean). Apache comes close, at 20ms slower median, 40ms slower mean. With no caching, the Dummy proxy is approximately 200ms slower. DeleGate, meanwhile, comes up last in mean response time, but almost 120ms faster than the Dummy for median.

DeleGate also shows a significant discontinuity in the latency distribution around 1 second in Figure 7. Intuition suggested that some kind of timeout was likely involved, which we were able to confirm upon examination of DeleGate's source code where we found the use of `poll(2)` with a timeout of 1000ms.

8.2 A reverse proxy workload

We also wish to be able to test content-based prefetching proxies. To avoid the need to interfere with an existing user base and systems, we decided to replay a Web server trace. Since fetching real content can have side effects [19], we specifically chose to use a log from a Web server that predominantly generated static content.

Thus, in this test we aim to measure the advantage (if any) that a content-based prefetching proxy can provide over a non-caching proxy. Instead of an artificial workload, we replayed the

[4]We have replicated these bugs (using a simple CGI and workload generator which are described at `http://www.cs.rutgers.edu/~davison/error/`) and have reported them to the maintainers of Apache and DeleGate.

origin server trace using the `simclient` tool from the Proxicizer toolset[5] since it was able to replay the logs in close to real time, preserving the inter-request delays that were originally present. The inter-request delays correspond to user "thinking times" between page requests and so retaining them is important in testing prefetching proxies so that the time available (between requests) to prefetch content matches what the proxy would see in a real-world situation.

8.2.1 Workload details

We started with the access logs from the Apache [7] Web server that runs our primary departmental web server, `www.cs.rutgers.edu`, for the month of May 2001 (almost one million requests, and only a few weeks old when these tests were performed). We filtered the original logs for malformed requests as well as requests for dynamic resources (using the heuristic that looks for URLs containing "cgi-bin", "?", or ".cgi", or use of the POST method). The remaining GET and HEAD requests to resources believed to be static numbered over 982k.

Since the simclient tool from the Proxycizer suite required logs in some proxy format, we converted the Apache logs to Squid logs. Over 50,000 distinct client IPs were seen. Surprisingly, 5.3% of those were unresolvable (but had been resolved when logged), and so we gave those hostnames addresses within the 192.168.X.X private IP address space.

Unfortunately, the Apache logs had only a one second timestamp resolution, so even though Squid logs typically have millisecond resolution, these logs would not. These logs did, however, have multiple requests at exactly the same time, which is extremely unlikely with a single processor under relatively light load. Thus, we arbitrarily spaced requests that had identical timestamps 10ms apart in the converted trace.

From this log we selected a subset of three days (May 2-4) to be replayed through the ROPE infrastructure. This represented just over 100,000 requests from 8095 distinct client IP addresses, with a peak request rate of 5 requests/sec.

A total of 746 robot IPs could be identified by looking for requests to `/robots.txt` in the full trace. We did not filter these clients in the three day test set, representing 14.4% of the requests. We did not analyze the effects that these robots had on the caching performance, but Almeida *at al.* [6] have found robot activity to have a significant impact.

We also examined the trace to determine the infinite cache size to help us size the caches we wish to test. If we sum the maximum size object returned for a particular cacheable URL, we get approximately 367MB. Unfortunately, we cannot measure the infinite cache size perfectly, because some responses change size over the trace, and the cacheability of some negative responses is under some debate.[6] Likewise, the sizes of HTTP/1.1 responses for objects with partial content give an inaccurate picture of the size of the whole object.

Even though the requests in the workload included many HTTP/1.1 requests, the simclient only generates HTTP/1.0 requests. This, combined with a slow connection-closing bug in an old version of the TIS firewall proxy under HTTP/1.1 propelled us to restrict DeleGate to generate HTTP/1.0 requests (since the other proxies did not support generating HTTP/1.1 requests anyway).

Unlike the original trace, the Proxycizer simclient does not generate IMS queries, since it does not have a cache of its own, nor does it know which of the original requests carried IMS (although we know some of them did, generating 304 responses).

[5] Proxycizer [22] is a suite of applications and C++ classes that can be used in simulating and/or driving web proxies [23].

[6] The HTTP/1.1 specification [21] does not permit the caching of 404 responses, for example, but documentation in the Squid config file suggests that Squid does: "Certain types of failures (such as "connection refused" and "404 Not Found") are negatively-cached for a configurable amount of time. The default is 5 minutes."

Proxy name	Relative BW-sent	Relative BW-rec	Responses delivered	% responses match direct	% responses not matching	Mean latency	Median latency
(direct)	100%	100%	73355	100.00%	0.00%	1428ms	123ms
Squid	93.2%	28.8%	73265	99.96%	0.04%	738	50
DeleGate	93.6%	17.4%	73187	99.57%	0.43%	893	82
Wcol	76.0%	202%	58023	97.33%	2.67%	1233	122
Dummy	92.8%	93.3%	72958	100.00%	0.00%	1361	223
Apache	93.8%	23.1%	73007	99.95%	0.05%	725	46

Table 2: Reverse proxy workload results.

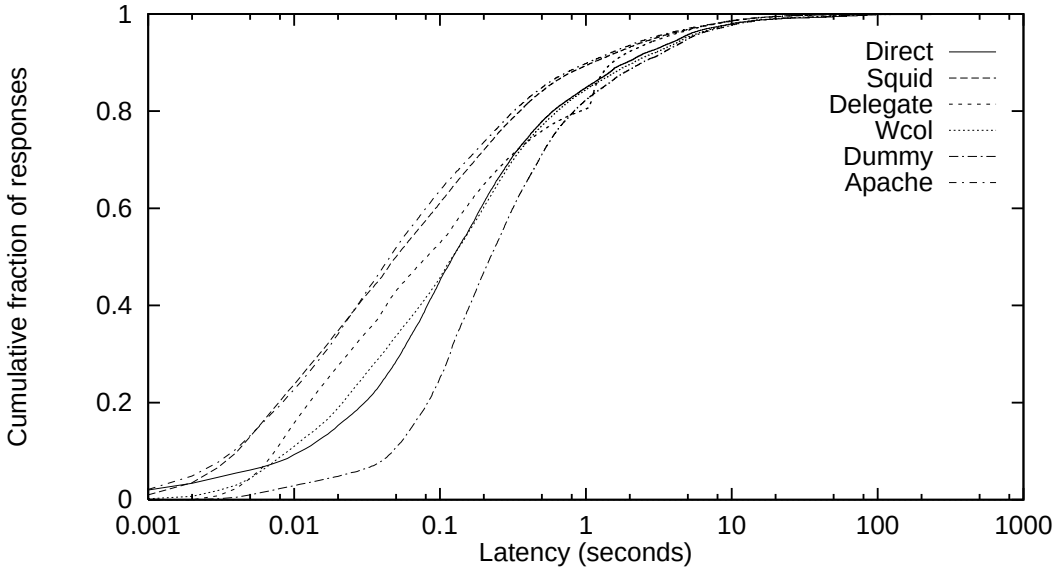

Figure 8: The cumulative distribution of response times for each proxy under the full-content reverse proxy workload.

Likewise, some clients (such as Adobe Acrobat [1]) generate HTTP/1.1 ranged requests which elicit HTTP/1.1 206 partial content responses. We are unable to model this well with the Proxycizer simclient, which only generates HTTP/1.0 requests, and so it instead generates multiple full requests when a real HTTP/1.0 client would have only made one such request (since the whole response would have been returned on the first request). Of the original three day log, 7.3% of the responses were for partial content (HTTP response code 206).

8.2.2 Results

Results from this experiment can be found in Table 2 and are graphically displayed in Figure 8. Apache does very well in this test, squeezing past Squid in all measures except the number of responses served. For some (currently) unknown reason, none of the tested proxies were able to generate even 94% of the response bytes. Possible explanations include the potential for large-sized responses to be among the hundreds that failed to be served, and the indication in logs that the Collector crashed twice during the run (automatically restarting).

Overall, DeleGate did reasonably well, requiring the least amount of bandwidth by far. Squid used more bandwidth than Apache, and over 65% more than DeleGate. Perhaps Squid is the most

conservative in cacheability, as its error rate is lowest. As in the previous experiment, DeleGate still shows (albeit less pronounced) the non-linearity at one second response times.

In this test, Wcol had the most failures to generate content that matched the direct connection. One reason we found was that Wcol interprets HEAD requests as GET requests, and passes back a body. Another possible source of problems was that Wcol does not generate a `Host:` header, which while not strictly part of the HTTP/1.0 specification, is a common extension (required in HTTP/1.1) that is needed to distinguish among virtual hosts using the same IP address. Wcol also crashed late in the test, and thus was unable to respond to approximately 20% of the requests. Thus, we would expect the bandwidth used by Wcol to be closer to 250% if it had finished the run.

9 Discussion/Future Work

Unfortunately, because of various limitations, the sample experiments reported here were unable to demonstrate improved performance from the use of prefetching. These limitations include:

- The use of an old and somewhat buggy prefetching proxy cache. We used WcolD because WcolE was last updated years ago and is still in beta (and was not easily compiled). Thus it is likely that bugs and inefficiencies that have been fixed in WcolE have affected us here.
- Replaying Web access logs (and using an artificial load generator). While the artificial load was useful to find some proxy bugs, replaying access logs is always limited. The Proxycizer simclient ran on a single machine, and thus likely had resource constraints not present in the original workload. Likewise, the recorded timestamps were unlikely to mirror well real-world inter-request timings. Finally, even though the trace was just a month old, the content referenced by it may have changed or moved.
- Use of a local host as target. We replayed the request log of a LAN-attached server, and thus requests and responses did not have to traverse the Internet. With the higher response times typical of the real Internet, caching and prefetching may be able to show greater effects.

The experience described in this status report has helped us think about what changes we would like to see next. These changes fall into three categories: experimental design, code implementation, and architecture. Under experimental design, we would like:

- To use a real live user load rather than replaying a trace.
- To add latency and bandwidth limits between collector and origin server, such as by using Squid delay pools (which would help with bandwidth, but not latency), or better, by fetching data from servers across the Internet.
- To use switched rather than a single shared ethernet between all systems to reduce the potential for interference.

In code implementation, we would like:

- To stress test the implementation to find throughput and connection limitations.
- To test a working multi-threaded implementation and compare to the multi-process version.

- To record connection establishment costs on a persistent per-host basis so that the costs can be used for subsequent simulated misses that require a new connection. (Even when this is performed, we realize that fake connection times do not necessarily reflect non-ROPE conditions, since network and origin server conditions vary over time).
- To mark with minimum freshness time all uncacheable responses and to throw away any prefetched uncacheable responses (since that is what the tested proxies should do with an uncacheable response). Likewise, it should enforce the minimum freshness time for responses with shorter (or non-existent!) expiration times.
- To allow the Multiplier to send requests to the tested proxies before receiving a reply from the Collector. This would allow the possibility of allowing the first response to be served to the client to speed use (but does open the possibility for other problems).
- To expire all content in the Collector after a short period of time since the its cache is only needed (or wanted) to prevent multiple (nearly) simultaneous requests.
- To restrict the Collector to in-memory cache only if possible to minimize the cost and variance of retrieving objects from the Collector.
- To dynamically adjust the delay imposed on subsequent requests for the same object to account for variance in retrieving object from disk or memory.
- To modify TCP drivers to support more accurate replication of connection setup times and data transfer timings.

Finally, in the architecture design we would like:

- To test "transparent" proxies under a similar architecture. The current version requires explicit configuration to use a parent proxy.
- To consider the effects of significantly differing connection timeouts and replacement policies, as connection management policies will affect proxy performance [15].
- To add management of DNS lookups. In the current implementation the first proxy doing the lookup will pay the penalty for a DNS miss, but not subsequent proxies, thus penalizing prefetching proxies inappropriately. Since DNS resolution costs can be severe [14], this will be an important effect to control.

We have outlined many areas above for future work. Some of these are straightforward, and their presence is just a matter of hindsight. Others, however, are more of a challenge, and may require significant effort.

10 Summary

This technical report has outlined the SPE architecture and elaborated on the current status of ROPE, the Rutgers Online Proxy Evaluator, our prototype implementation of SPE, and many of the issues related to it. We have described validation tests that we have performed on ROPE, as well as demonstrated the use of the current prototype in two experiments to simultaneously evaluate up to four publicly available proxy cache implementations. As expected, we measured bandwidth used and response latencies, but also discovered unexpected caching bugs in two of the proxies tested. Difficulties with the only prefetching proxy cache prevent us from reaching any conclusions with the use of prefetching to improving performance.

Acknowledgments

This work has been supported by the National Science Foundation under grant NSF ANI 9903052. We thank Vincenzo Liberatore, Weisong Shi, and Geoff Voelker for valuable comments on an earlier draft. We also thank the folks at the Measurement Factory for permission to publish comparative results based on Polygraph-generated workloads.

References

[1] Adobe Systems Incorporated. Adobe Acrobat Reader. `http://www.adobe.com/products/acrobat/readermain.html`, 2001.

[2] J. Almeida, V. A. F. Almeida, and D. J. Yates. Measuring the behavior of a World Wide Web server. In *Proceedings of the Seventh Conference on High Performance Networking (HPN)*, pages 57–72. IFIP, Apr. 1997.

[3] J. Almeida, V. A. F. Almeida, and D. J. Yates. WebMonitor: a tool for measuring World Wide Web server performance. *first monday*, 2(7), July 1997.

[4] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. *Computer Networks And ISDN Systems*, 30(22-23):2179–2192, Nov. 1998. Proc. of the Third Web Caching Workshop, June 1998.

[5] J. Almeida and P. Cao. Wisconsin proxy benchmark 1.0. Available from `http://www.cs.wisc.edu/~cao/wpb1.0.html`, 1998.

[6] V. Almeida, D. Menascé, R. Riedi, F. Peligrinelli, R. C. Fonseca, and W. Meira, Jr. Analyzing the impact of robots on performance of Web caching systems. In *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, Boston, MA, June 2001.

[7] Apache Group. Apache HTTP server documentation. Available at `http://httpd.apache.org/docs/`, 2001.

[8] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal*, 2(1-2):69–83, 1999. Special Issue on World Wide Web Characterization and Performance Evaluation.

[9] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 151–160, Madison, WI, June 1998.

[10] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[11] T. Berners-Lee, R. T. Fielding, and H. Frystyk Nielson. Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, `http://ftp.isi.edu/in-notes/rfc1945.txt`, May 1996.

[12] CacheFlow Inc. Active caching technology. `http://www.cacheflow.com/technology/whitepapers/active.cfm`, 2001.

[13] K.-i. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[14] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing Web latency. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, Mar. 2000.

[15] E. Cohen, H. Kaplan, and J. D. Oldham. Managing TCP connections under persistent HTTP. *Computer Networks*, 31:1709–1723, 1999.

[16] B. D. Davison. A Survey of Proxy Cache Evaluation Techniques. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 67–77, San Diego, CA, Mar. 1999.

[17] B. D. Davison. Measuring the Performance of Prefetching Proxy Caches. Poster presented at the ACM International Student Research Competition, New Orleans, March 24-28, 1999 and at the AT&T Student Research Symposium (a regional ACM Student Research Competition), November 13, 1998. Awarded third place and first place respectively., 1999.

[18] B. D. Davison. Simultaneous Proxy Evaluation. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 170–178, San Diego, CA, Mar. 1999.

[19] B. D. Davison. Assertion: Prefetching with GET is not good. In *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, Boston, MA, June 2001.

[20] B. D. Davison. web-caching.com: Web caching and content delivery resources. `http://www.web-caching.com/`, 2001.

[21] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, `http://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[22] S. Gadde. The Proxycizer Web proxy tool suite. Available at `http://www.cs.duke.edu/ari/cisi/Proxycizer/`, 2001.

[23] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[24] T. Johnson. Webjamma. Available at `http://www.cs.vt.edu/~chitra/webjamma.html`, 1998.

[25] Junkbusters Corporation. The Internet Junkbuster proxy. Available from `http://www.junkbuster.com/ijb.html`, 2000.

[26] M. Koletsou and G. M. Voelker. The Medusa proxy: A tool for exploring user-perceived Web performance. In *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, Boston, MA, June 2001.

[27] R. Liston and E. Zegura. Using a proxy to measure cleint-side web performance. In *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, Boston, MA, June 2001.

[28] S. Manley, M. Seltzer, and M. Courage. A self-scaling and self-configuring benchmark for Web servers. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 270–271, Madison, WI, June 1998.

[29] D. Mosberger and T. Jin. httperf—A tool for measuring Web server performance. *Performance Evaluation Review*, 26(3):31–37, Dec. 1998. Workshop on Internet Server Performance (WISP'98).

[30] R. Rivest. The MD5 message-digest algorithm. RFC 1321, `http://ftp.isi.edu/in-notes/rfc1321.txt`, Apr. 1992.

[31] A. Rousskov. Web Polygraph: Proxy performance benchmark. Available at `http://www.web-polygraph.org/`, 2001.

[32] A. Rousskov and D. Wessels. The third cache-off. Raw data and independent analysis at `http://www.measurement-factory.com/results/`, Oct. 2000.

[33] Y. Sato. DeleGate home page. Online at `http://www.delegate.org/`, 2001.

[34] D. Wessels. Squid Web proxy cache. Available at `http://www.squid-cache.org/`, 2001.