In *Proceedings of the 30th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG)*, Las Vegas, NV, December 2004.

IMPLEMENTING A WEB PROXY EVALUATION ARCHITECTURE

Brian D. Davison and Baoning Wu
Department of Computer Science & Engineering, Lehigh University
19 Memorial Drive West, Bethlehem, PA 18015 USA
{davison,baw4}@cse.lehigh.edu

The evaluation of Web proxy performance can be complex. In this paper, we present the lessons and results of our implementation of a novel simultaneous proxy evaluation technique. In this architecture, we send the same requests to several proxies simultaneously and then measure their performance (e.g., response times). With this approach, we can test proxies using a live network with real content, and additionally be able to evaluate prefetching proxies.

1 Introduction

A caching web proxy is used to store cachable web content in the expectation that it will be of use in the future. If a request for a Web object arrives while the cached copy is still valid, the object will be served from the cache instead of from the origin server. Both response time and bandwidth can be saved by applying web proxies.

Proxy caches are useful and deployed across many organizations. It is helpful both to developers and to purchasers of web proxies to be able to accurately measure their performance. Typical evaluation methods, such as benchmark performance, however, are limited in applicability — often they are not representative of the traffic characteristics of any customer facility.

In contrast, the Simultaneous Proxy Evaluation (SPE) architecture [9] is designed specifically to use actual client requests and an existing network connection to evaluate proxy performance. To enforce comparability of proxy measurements, it evaluates proxies in parallel on the same workload.

Our implementation of SPE, called LEAP (Lehigh Evaluation Architecture for Proxies), can be used to measure the average response time, hit ratio, and consistency of the data served by a proxy. Compared to our earlier SPE implementation work [9, 12], LEAP is more robust and accounts for more variables that affect measurement accuracy.

Building a complex system to perform accurate evaluation has provided some challenges. In what follows, we first provide some details about SPE and our implementations. We then review a number of design challenges that we had to face, and our solutions to them and lessons learned from them. We subsequently describe validation tests of our system, and demonstrate the use of LEAP by measuring performance characteristics of multiple open-source web proxies by replaying requests from a real-world proxy log. We wrap up with related work, discussion, and a summary.

This material is based upon work supported by the National Science Foundation under Grant No. ANI 9903052. Any opinions, findings, and conclusions for recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2 The SPE Architecture

The Simultaneous Proxy Evaluation (SPE) architecture [9] is designed to permit a live workload and the existing network connection to fairly evaluate multiple proxies. LEAP is our implementation of this architecture. It records timing measurements to calculate proxy response times and can compute page and byte hit rates. In summary, it forms a wrapper around the proxies being tested and produces logs that measure external network usage as well as performance and consistency as seen by a client. By simultaneously evaluating multiple proxies, we can utilize a single, possibly live, source of Web requests and a single network connection to provide objective measurements under a real load. Importantly, it also allows for the evaluation of content-based prefetching proxies, and can test for cache consistency. In this section, we introduce its architecture (illustrated in Figure 1).

Accompanied by a few simple log analysis tools, the *Multiplier* and *Collector* are the two major pieces of software that make up any SPE implementation. In LEAP, the Multiplier is a stand-alone multi-threaded program which is compatible with HTTP/1.1 [14] and HTTP/1.0 [7] protocols and was developed from scratch. The publicly available Squid proxy cache [29] version 2.5PRE7 was modified extensively to develop the Collector.

In the SPE architecture, clients are configured to con-

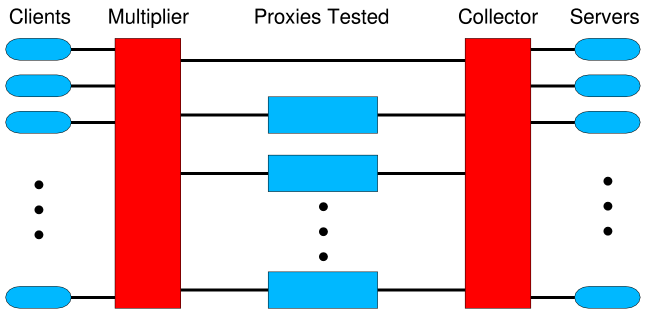


Figure 1: The SPE architecture for the simultaneous online evaluation of multiple proxy caches.

nect to the Multiplier.[1] Requests received by the Multiplier are sent to each of the tested proxies. Each attempts to satisfy the request, either by returning a cached copy of the requested object, or by forwarding the request to a parent proxy cache, the Collector. The Collector then sends a single copy of the request to the origin server for fulfillment when necessary (or potentially asks yet another upstream proxy cache for the document). When the Multiplier gets responses from the tested proxies, it will record the response time and calculate an MD5 digest for each response from each tested proxy. After the experiment, Perl scripts are used to analyze the logs from the Multiplier and the performance reports are generated for these tested proxies.

2.1 Architecture Details

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.

Each of the proxy caches being evaluated can be treated as a black-box — we do not need to know how they work, as long as they function as a valid HTTP/1.0 or 1.1 proxy cache. This is helpful in particular for commercial proxies, but in general eliminates the requirement for either detailed knowledge of the algorithms used or source code which is needed for simulation and specialized code additions for logging [21] respectively. Typically the proxy caches would run on different machines, but for simple tests that may not be necessary. Alternatively, the same software with different hardware configurations (or vice versa) can be tested in this architecture.

In order to prevent each proxy cache from issuing its own request to the destination Web server, the Collector functions as a cache to eliminate extra traffic that would otherwise be generated. It cannot be a stan-

[1]We assume an explicit client confi guration, but SPE could also work in a transparent proxy confi guration.

Sample HTTP/1.1 request:
```
GET http://news.bbc.co.uk/shared/img/logo04.gif HTTP/1.1
Host: news.bbc.co.uk
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;
rv:1.6) Gecko/20040510
Accept: image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://news.bbc.co.uk/
```

Sample HTTP/1.0 response:
```
HTTP/1.0 200 OK
Date: Wed, 23 Jun 2004 20:52:01 GMT
Server: Apache
Cache-Control: max-age=63072000
Expires: Fri, 23 Jun 2006 20:52:01 GMT
Last-Modified: Tue, 27 Apr 2004 09:19:35 GMT
ETag: "fb-408e25a7"
Accept-Ranges: bytes
Content-Length: 251
Content-Type: image/gif
Age: 455
X-Cache: HIT from wume1.local.cse.lehigh.edu
Proxy-Connection: keep-alive
```

Figure 2: Sample HTTP request from Mozilla and response via a Squid proxy.

dard cache, as it must eliminate duplicate requests that are normally not cachable (such as those resulting from POSTs, generated by cgi-bin scripts, designated as private, etc.). By caching even uncachable requests, the Collector will be able to prevent multiple requests from being issued for requests that are likely to have side effects such as adding something to an electronic shopping cart. However, the Collector must then be able to distinguish between requests for the same URL with differing arguments (i.e., differing fields from otherwise identical POST or GET requests).

2.2 The Multiplier in LEAP

The Multiplier listens on a configurable port for TCP connections from clients and assigns a thread from a pre-generated pool for every client connection opened. Connections to the tested proxies are handled by this thread using `select(2)`. When a complete request is received, 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) from the client and responses from the proxies.

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

```
Wed Apr 28 20:29:16 2004:(id=18972:2761927472):LEAPRESULT: Collector:  0:237527 0:725 2248 {DUMMY,} 192.168.0.101
non-validation HTTP/1.1 GET http://www.google.com/ HTTP/1.1
Wed Apr 28 20:29:16 2004:(id=18972:2761927472):LEAPRESULT: Proxy-1:  0:313149 0:372 2248 {200,200,OK,} 192.168.0.101
non-validation HTTP/1.1 GET http://www.google.com/ HTTP/1.1
Wed Apr 28 20:29:16 2004:(id=18972:2761927472):LEAPRESULT: Proxy-2:  0:432901 0:291 2248 {200,200,OK,} 192.168.0.101
non-validation HTTP/1.1 GET http://www.google.com/ HTTP/1.1
Wed Apr 28 20:29:16 2004:(id=18972:2761927472):LEAPRESULT: Proxy-3:  0:251789 0:453 2248 {200,200,OK,} 192.168.0.101
non-validation HTTP/1.1 GET http://www.google.com/ HTTP/1.1
```

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)and BODY_MISMATCH (if the response body differs).

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 factors like the time to establish a connection, the size of object retrieved, the initial latency, the total time to transfer the entire object and a signature (using MD5 checksums [25]) of the object. A data structure with this information is maintained and once all the responses for this request have been received, 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 signatures of the responses sent by collector and each test proxy. Mismatches, if any, are recorded. Figure 3 shows sample lines from a Multiplier log.

2.3 The Collector in LEAP

In this work, we have made changes only to the source code of Squid to implement the Collector within LEAP. We have not modified the underlying operating system.

Recall that the Collector is a proxy cache with additional functionality:

- *It will replicate 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 uncachable responses.* This prevents multiple copies of uncachable responses from being generated when each tested proxy makes the same request.

The basic approach used in our system is as follows: if the object requested by a client is not present in the cache, we fetch it from the origin server. We record the DNS resolution time, connection setup time, response time and transfer time from the server for this object. For the subsequent requests from clients for the same object in which the object is in the cache, we incorporate an artificial delay so that even though this request is a cache hit, the total response time experienced is the same as a cache miss.

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 by using Squid's built-in event-driven scheduling apparatus. Instead of sending the chunk right away, we schedule the execution of a new function to send the chunk at the desired time [13].

To cache uncachable objects in Squid, two modifications are required. The algorithm which determines the cachability 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 normally be treated as uncachable. Squid's `httpMakePrivate()` function in `http.c` marks an object as uncachable 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 180 seconds. This minimum *freshness time* was chosen to be long enough that requests from all tested proxies should be received within this time period. Thus, subsequent requests for this (otherwise uncachable) object are serviced from the cache if they are received before the object expires from the cache.

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).

```
1083714572.228 394 192.168.0.2 TCP_MISS/200 3108 GET
http://www.pinkforfree.com/images/default/button_members_off.jpg - DIRECT/208.236.11.57 image/jpeg

1083714577.243 406 192.168.0.201 TCP_MEM_HIT/200 3112 GET
http://www.pinkforfree.com/images/default/button_members_off.jpg - NONE/- image/jpeg

1083714640.398 416 192.168.0.2 TCP_MISS/200 11325 GET
http://media2.travelzoo.com/msn/ - DIRECT/64.56.194.107 text/html

1083714645.404 413 192.168.0.201 TCP_HIT/200 11329 GET
http://media2.travelzoo.com/msn/ - 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.

3 Lessons from building LEAP

Carefully implementing the SPE architecture has provided us with a number of interesting design challenges. We describe them here, along with the solutions that we applied.

3.1 Separate DNS cost

Issue: In order for a second instance of the same request to be delayed correctly, we need to record the initial response time, i.e., the time interval between when the request is sent out to origin server and when the first packet of response from origin server comes back. At first, we recorded this time interval as a whole, but this generated unsatisfactory timing results.

One potentially major time cost for getting response is the DNS resolution time for getting the IP address of origin server. This is especially true for cases of DNS failures (no response from DNS server, etc.). In our tests, less than .7% of requests had DNS failures, of those that did, approximately 25% had DNS failures in which responses timed out after more than 100 seconds. In general, we don't want to force a proxy to wait for the DNS costs of resolving a host name if it has already done so, by previously requesting pages from the same origin server. Therefore, we need to record the DNS resolution time for each origin server by each tested proxy separately.

Implementation: Our final solution is to record the DNS resolution time separately and we also record which tested proxy has paid this resolution time.

A standard Squid uses the `_ipcache_entry` structure to implement the DNS callback mechanism. We added several new member fields to this structure; the most important fields are `request_time`, `firstcallbacktime` and `address`. The `request_time` is used to record when the DNS request is sent out for resolution, and the `firstcallbacktime` is used to record the first time the DNS answers comes back and this structure is used. We use the time difference as the DNS time for this object. The `address` field is used to record the IP address of tested proxies which have performed this DNS resolution (that is, they paid the cost of a lookup).

One problem is that sometimes several tested proxies send same requests and they all wait for the DNS resolution process. So the following proxies shouldn't wait the same as the first one which forces Squid to do DNS resolution. We have a function named `dnssleep_time()`, that will send back the proper time value for each different proxy's DNS time. The basic idea within the `dnssleep_time()` function is that we deduct the time that has passed since the proxy has begun waiting for DNS resolution from the total DNS time that is recorded in the `_ipcache_entry` structure, which is the first DNS resolution result for this DNS request.

Lesson: DNS delays are potentially significant, so they cannot be ignored.

3.2 Who gets the response first?

Issue: Since we modified Squid proxy to get our Collector, we have some restrictions from the basic Squid mechanism.

We observed at first that some subsequent responses were served much faster than they should be although we recorded the time interval correctly and usually this happened if the response contains only one packet. After careful debugging, the cause of this problem is that sometimes it is possible that one certain request comes from tested proxy 1 to the Collector and later the same request comes from tested proxy 2, but the response from origin server still hasn't come yet, so both these two requests are waiting for the same response. We also found that either of these requests can get the first packet of the response according to the implementation of Squid proxy. So, if the request from proxy 2 gets the response first and then gets served earlier than the request from proxy 1, it is not surprising that the proxy 2 will get the completed response faster than proxy 1. Finally, if the response is only one packet, surely proxy 2 will get the whole response earlier than proxy 1 since proxy 2 is not penalized at all.

Implementation: It is quite complicated to change the Squid mechanism, so our solution is to add a new flag `ifsentout` in the `_clientHttpRequest` structure.

We set this flag to 1 if the request is really sent out to origin server and 0 otherwise. For the above case, the `ifsentout` will be set to 1 for the request from proxy 1 and 0 for the request from proxy 2. Then we can use this flag together with the `newtotal` variable in `_StoreEntry` to decide if the response should be sent out. We will send out the response either the `ifsentout` is 1 (meaning that this is the proxy that first issued this request) or `ifsentout` is 0 but `newtotal` has a valid time value (meaning that the result has already been sent to the first proxy).

Lesson: Don't assume that an event-driven server will serve pending requests in any particular order.

3.3 HTTP If-Modified-Since request

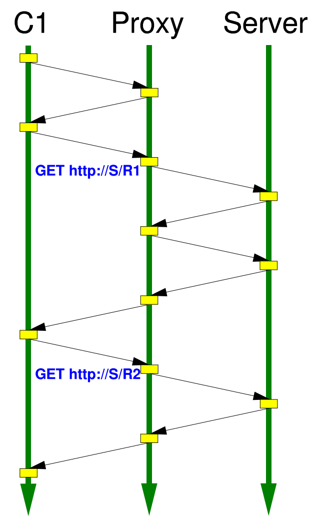
Issue: Sometimes, If-Modified-Since requests will be sent by proxies. Previously we have considered only the case in which a client makes a request and either the proxy has a valid copy of the requested object or it does not. In reality, it can be more complicated. If the client requests object R, and the cache has an old version of R, it must check to see if a modified version is available from the origin server. If a newer one exists, then we have the same case as a standard cache miss. If the cached version is still valid, then the proxy has spent time communicating with the origin server that must be accounted for in the delays seen by the client. The old time interval for delaying subsequent requests is not accurate in this situation and must be updated.

Implementation: The general principle we use is to reproduce any communication delays that would have occurred when communicating with the origin server. Thus, if the proxy does communicate with the origin server, then the connection establishment and initial response time delay including the DNS resolution needs to be updated. If the proxy is serving content that was cached, then the transfer time delay (as experienced when the content was received) is kept unchanged. If new content is sent from origin server, then the transfer time delay is also updated. Here the flag `ifsentout` in the `_clientHttpRequest` is used to decide if these time intervals need to be updated.

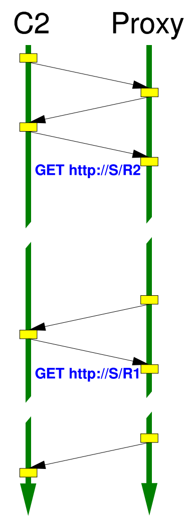
Lesson: Be sure to consider all possible combinations of inputs.

3.4 Persistent connections

Issue: Most Web clients, proxies and servers support persistent connections (which are a commonly implemented extension to HTTP/1.0 and the default in HTTP/1.1). Persistent connections allow subsequent requests to the same server to re-use an existing connection to that server, obviating the TCP connection es-



(a) Client re-uses persistent connection.



(b) Second client finds cached data.

Figure 5: Transaction timelines showing how persistent connections complicate timing replication.

tablishment delay that would otherwise occur. Squid supports persistent connections between client and proxy and between proxy and server. This process is sometimes called *connection caching* [8], and is a source of difficulty for our task.

Consider the case in which client *C1* requests resource *R1* from origin server *S* via a proxy cache (see the transaction timeline in Figure 5a). Assuming that the proxy did not have a valid copy of *R1* in cache, it would establish a connection to *S* and request the object. When the request is complete, the proxy and server maintain the connection between them. A short time later, *C1* requests resource *R2* from the same server. Again assuming the proxy does not have the resource, the connection would be re-used to deliver the request to and response from *S*. This arrangement has saved the time it would take to establish a new connection between the proxy and server from the total response time for *R2*.

Assume, however, a short time later, client *C2* also requests resource *R2*, which is now cached (Figure 5b). Under our goals, we would want the proxy to delay serving *R2* as if it were a miss. But a miss under what circumstances? If it were served as the miss had been served to *C1*, then the total response time would be the sum of the server's initial response time and transfer time when retrieving *R2*. But if *C1* had never made these requests, then a persistent connection would not exist, and so the proxy would have indeed experienced the connection establishment delay. Our most recent measurement of that time was from *R1*, so it could be re-used. On the other hand, if *C2* had earlier made a request to *S*, a persistent connection might be warranted. Similarly, if *C2* were then to request *R1*, we would not want to replicate the delay that was experienced the first time *R1* was retrieved, as it included the connection establishment time.

In general it will be impossible for the Collector to determine whether a new request from a tested proxy would have traveled on an existing, idle connection to the origin server. The existence of a persistent connection is a function of the policies and resources on both ends. The Collector does not know the idle timeouts of either the tested proxy nor the origin server. It also does not know what restrictions might be in place for the number of idle or, more generally, simultaneous connections permitted.

Implementation: While an ideal LEAP implementation would record connection establishment time separately from initial response times and transfer times, and apply connection establishment time when persistent connections would be unavailable, such an approach is not possible (as explained above). Two simplifications were possible — to simulate some policies and resources on the proxy and a fixed maximum idle connection time for the server side, or to serve every response as if it were a connection miss. For this implementation, we chose the latter, as the former would require the maintenance of new data structures on a per-proxy-and-origin-server basis, as well as the design and simulation of policies and resources.

The remaining decision is whether to use persistent connections internally to the origin servers from the Collector. While a Collector built on Squid (as ours is) has the mechanisms to use persistent connections, idle connections to the origin server consume resources (at both ends), and persistent connections may skew the transfer performance of subsequent responses as they will benefit from an established transfer rate and reduced effects of TCP slow-start.

Therefore, in our implementation, we modified Squid to never re-use a connection to an origin server. This effectively allows us to serve every response as a connection miss, since the timings we log correspond to connection misses, and accurately represent data transfer times as only one transfer occurs per connection.

Lesson: Sometimes it is better to control a variable than to model it.

3.5 Inner Collector Re-forwarding

Issue: Below, in the Validation section, we discuss testing LEAP within LEAP. Normally we would use the same code for both the inner Collector and the outer Collector. Since there should be some delay for the path through the inner LEAP, the average response time for the inner LEAP is bigger than the direct path. But sometimes we found the difference between these two average response time is abnormally high, especially in a workload which generates many errors. After careful debugging, we found that for certain requests, the outer Collector will attempt to contact the origin servers, fail, and send back one of a set of problematic response codes to the tested proxies. For a particular set of error codes (shown in Table 1), Squid developers apparently chose to attempt a direct connection (even when the configuration specifies the use of an upstream proxy only) rather than send back the errors generated by an upstream proxy. As a result, when Squid receives such an error code as a result of a failure, it will make the same attempt again. As a result, our version of Squid that is running in the inner LEAP will end up paying expensive delays twice.

HTTP_FORBIDDEN: (403)
HTTP_INTERNAL_SERVER_ERROR: (500)
HTTP_NOT_IMPLEMENTED: (501)
HTTP_BAD_GATEWAY: (502)
HTTP_SERVICE_UNAVAILABLE: (503)
HTTP_GATEWAY_TIMEOUT: (504)

Table 1: HTTP error codes for which Squid will attempt a direct connection instead of passing the error on to the client.

Implementation: We decide to use different code for the inner Collector and the outer Collector for the LEAP within LEAP test. For the outer Collector, we still use its original mechanism for deciding whether to re-forward requests. But for the inner Collector, we change the `fwdReforwardableStatus(http_status s)` function in `forward.c` file to let it always return 0 no matter what the status `s` is. Then for the inner Collector, it won't re-forward requests to origin server or other peers after receiving problematic response codes from the outer Collector.

Lesson: The good intentions of developers (to make their server robust) can sometimes get in the way.

3.6 Thread Pool

Issue: The Multiplier uses a distinct thread to handle each request from the client. Originally we did not create a new thread until we got a new request from a client. Since the creation of a thread takes some time, this time interval will add to the client's perceived delay.

Implementation: We use a thread pool instead in the Multiplier. The thread pool is built during the initialization process of the Multiplier. Each thread has a thread ID and there is an array `busy` with one element for each thread to show whether this thread is currently busy or not. When a request comes from a client, we will check which thread in the pool is free by checking the `busy` array. If we find one, we will pass the HTTP request and the file descriptor as arguments to the thread routine.

Lesson: For best performance, pre-allocate and/or pre-execute code so that it is performed in advance of a user request.

3.7 Incremental signatures

Issue: One of the functionals of the Multiplier is to calculate a signature for each response from the Collector and tested proxies. Then it will compare the signature from the Collector and those from the tested proxies to see whether the responses are different. We used an open-source implementation of MD5 (`md5.c` from RSA Data Security, Inc.) in LEAP. Initially we held the response in memory until we received the whole response and then we calculated the MD5 digest and freed the memory. Since we don't need to hold the response for future use in the Multiplier, holding the entire response, even temporarily, wastes a lot of memory, particularly during intensive tests with many active threads.

Implementation: We changed the code of `md5.c` so that the calculation can be performed incrementally, i.e., for each arriving part of the response, we calculate the partial digest and store the partial digest `MD5_CTX *context` in a `resInfoPtr` structure which is used for recording information for one request. After each calculation, only the partial MD5 digest is stored and the partial response is freed. One trick here is that MD5 digest uses 64 bytes as a basic block and it will pad the last block if the length is less than 64. For each partial response, the length may not be a multiple of 64, so we need to record the last part of this partial response if the length of it is less than 64 and add to the head of next partial response before we divide the next partial response into 64 bytes blocks. To achieve this goal, we modified original `md5.c` file, and we also have a special argument `type` passed to `MD5Update()`function to tell whether this is the last chunk of the response, if so and the last block has less than 64 bytes, `MD5Update()` will pad the last block, otherwise it will return the last block for future calculation.

Lesson: Incremental processing is often more efficient.

3.8 Event Driven Multiplier

Issue: In prior versions, no event-driven mechanism was used in the Multiplier. Once the response from the Collector through the direct connections comes back, all copies of the request to the different tested proxies were sent out sequentially. In addition, since the handling of a single request was single-threaded, we used non-blocking connections to the tested proxies. As a result, it was quite possible that the connection hasn't been established when we attempt to send the requests. If this happened, we would try a second time, and then give up. As a result, we would have more error cases, especially with higher workloads.

Implementation: Motivated by the event-driven scheduler in Squid, we find it effective to address the above issue by using a similar approach in the Multiplier.

A new structure, `ev_entry`, is defined to record the event information. Each object of this structure corresponds to sending a request to a proxy. Since there may be several tested proxies in the whole testing environment, an event list is formed by linking each event object. During the each run of the main loop, the event list is checked to see if there are any event ready. If so, the event is executed. After sending the request to the tested proxy, if the running is successful, the event is moved from the eventlist, otherwise the event will be kept in the event list for the next loop. In order to prevent endless event, we set a timeout so that the event has to be moved from the event list if it still can not make a success after that timeout.

By using this event driven mechanism, we can also do some scheduling more flexibly. For example, instead of sending the same requests to all the tested proxies at the exactly same time (which can cause small but real delays from processor and network device contention), we set a small time interval between the sending of the request to each subsequent proxy. This partially solves the "Who gets the response first" problem described earlier because a response is likely to have at least started being received by the time the second proxy sends its request.

Lesson: In timing-sensitive applications, don't try to do everything all at once.

3.9 Chunked Coding Handling

Issue: Some tested proxies may send back chunked coding responses. Since we don't need to pass back the response from the tested proxies to the client, initially we did nothing to these responses. But one major function of the Multiplier is to calculate the signatures of the responses from the Collector and the tested proxies. If the signatures don't match, the Multiplier will record that one of the tested proxies sent back a different response. Initially we found that proxies like Apache [5] and Delegate [28] always have lots more mismatches than other proxies. In fact, most of the responses were identical except they were sent back in chunked-encoded format.

Implementation: We implemented the code to handle chunked coding response in the Multiplier. The function $process\_chunked\_code$ is called when the header of the response has “Transfer-Encoding: chunked”. Similar to the “Incremental signature” problem, the tricky part for chunked coding is that we need to store the partial result for the next packet of the response. For example, we may only get a portion of the length of the chunk this time, e.g., the length is “706” for the following chunk, but only “70” is transfered with this packet, and “6” is at the beginning of the next, so we need to store “70” until we get the next packet. In addition, the chunked coding function should be called first before the MD5 digest calculation.

Lesson: Beware of syntactically different formats of semantically equivalent data.

4 Experimental Validation

Since we have an implementation of LEAP, the natural question is whether the results from our system are accurate and reliable and what is the impact of our system to the tested proxies. In this section, we will show some experiments to evaluate our system.

4.1 Data set

We started with the NLANR IRCache request log from uc.us.ircache.net (located in Urbana-Champaign, Illinois) for 9 September 2004. We selected only those requests that were served from the cache with a 200 response code. In this way we attempted to ensure that the contents retrieved by our traces would be cachable and safe (free from unwanted side-effects [11]). Out of those 97,462 requests, 57,685 were unique, allowing tested proxy caches to potentially be able to serve approximately 40% from cache. We replayed these requests in the experimental measurements (Section 5), and a set with similar characteristics from 25 April 2004 for the validation tests.

4.2 Experimental Method

Unless described otherwise, for all experiments we used the same architecture as showed in Figure 1. The only difference is that we only use one client — httperf [23] — to send a sequence of requests at a given rate (typically 10 requests per second), using a separate connection per request, as we replay the proxy trace described above once. We run httperf on a one machine, from which it sends all requests to the Multiplier on another machine. Then the Multiplier will send requests to the tested proxies: one or more of Oops 1.5.23 [17], Squid 2.5.Stable4 [29], Apache 2.0.50 [5]

Connection type	mean	median
PT Proxy direct	550ms	54ms
PT Proxy tested	551ms	58ms

Table 2: Comparison of response times between direct and tested proxy connections.

Connection type	mean	median
direct	455ms	55ms
inner LEAP	470ms	52ms

Table 3: Evaluation of LEAP implementation overhead.

and a pass-through proxy implemented by our Multiplier. Each of these proxies runs on an identical machine and is configured to use the Collector (on an independent machine) as a parent proxy. So to test four proxies, altogether we would use seven machines. The caching proxies Squid, Apache and Oops are configured to use up to 8MB of in-memory cache, and 64MB of cache on disk (other settings are left at defaults).

4.3 Proxy penalty

We are first concerned with the internal architecture of our LEAP implementation. We want to know whether the implementation imposes extra costs for the proxies being tested, as compared to the direct connection between Multiplier and Collector. The experiment we chose is to force both the direct connection and tested proxy to go through identical processes. 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 with a portion of the IRCache workload we found that the direct connection and the connection through the Multiplier have similar response time on average. They are 550ms and 551ms respectively (shown in Table 2). From this we can tell our Multiplier adds a minimal delay to tested proxy measurements. That is, the overhead of the code for direct connections is quite similar to the overhead of handling the proxies.

4.4 Implementation overhead

Here we are concerned with the general overhead that a LEAP implementation introduces to the request/response data stream. We want to know how much worse performance will be for the users behind the 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 LEAP implementation using another LEAP implementation, which will be called LEAP

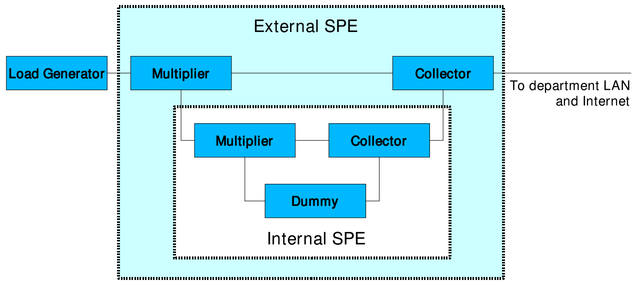


Figure 6: Configuration to test one LEAP implementation within another.

within LEAP in the rest of this paper, as shown in Figure 6. We again used the same IRCache artificial workload driven by httperf as input to the external LEAP. The inner LEAP (being measured) had just one PT Proxy to drive. The outer one measured just the inner one.

The results of our test are presented in Table 3. We found that the inner LEAP implementation generated a mean response time of 470ms, as compared to the direct connection response time of 455ms. Medians were smaller, at 55ms and 52ms, respectively. Thus, we estimate the overhead (for this configuration) to be approximately 15ms.

The cumulative fraction of responses distribution for the LEAP within LEAP testing is shown in Figure 7. The solid line is used to represent the direct connection and the dashed line is used for the inner LEAP. As we can see from this figure, these two lines are quite close to each other. This suggests that our LEAP architecture has relatively little impact on the real world from the client's view.

4.5 Fairness to tested proxies

Since we are measuring the performance of multiple proxies simultaneously, we wish to verify that we can do so fairly. That is, that each proxy is treated identically,

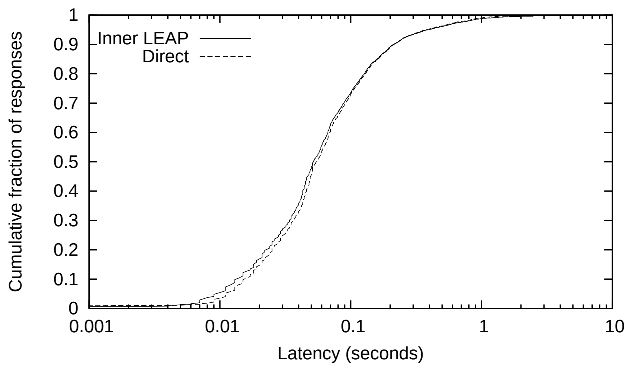


Figure 7: The cumulative distribution function of response times for LEAP within LEAP.

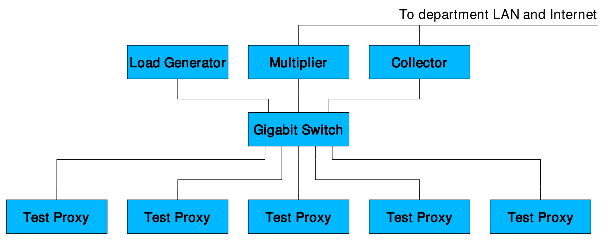


Figure 8: Experimental network configuration.

no matter what order they are in within the system. We verified this aspect of our system by testing four identical proxies — all running the same Squid configuration on machines with identical hardware. We found that measurements of mean and median response times would indeed vary between proxies by only a few milliseconds. Thus we conclude that we are able to fairly test the proxies.

5 Experimental Results

5.1 Network configuration

For each test we run a Multiplier and a Collector as described above. As shown in Figure 8, each proxy runs on a separate system, as do each of the monitoring and load-generating processes. Each machine running a proxy under test includes a 1Ghz Pentium III with 2GB RAM and a single 40GB IDE drive. All machines are connected via full-duplex 100Mbit Ethernet to a gigabit switch using private IP space. A standard version of Linux (kernel 2.4) was installed on each machine. The Multiplier and Collector ran on somewhat more powerful systems but were dual-homed on the private network and on the departmental LAN for external access.

5.2 Results

Four different proxies are used in the LEAP architecture: Oops, Apache, Squid, and a simple pass-through proxy (PT Proxy) which is implemented using our Multiplier, since with only one directed connection, it can be used as a proxy without cache. In this experiment, Oops required only 21.5% of the bandwidth of the pass-through proxy and Squid used 21.0%. In comparison, Apache was more agressive and cached more, using only 17.0% of the non-caching bandwidth.

From the results showed in Table 4, we can see Squid has the best mean and median response time. Apache has similar median response time and a slightly larger mean response time. Oops has larger mean and median than Squid and Apache, but smaller than PT Proxy. PT Proxy has similar mean and median response time as the Direct connection.

Proxy name	Mean time for response	Median time for response
Direct	491ms	42ms
Squid	410ms	35ms
Oops	459ms	41ms
PT Proxy	487ms	43ms
Apache	424ms	35ms

Table 4: Artificial workload results.

The cumulative fraction of responses is shown in Figure 9. From the figure, we can see that Apache and Squid, which are the first and second curve from the top, have similar distribution. Oops is worse than them, but better than PT Proxy, so it's curve is in the middle. The Collector and PT Proxy have similar response-time distributions.

Since Apache and Squid have caches, and there are repeated requests in our test data set, these two proxies can take advantage of serving responses from cache. Thus, their response-time distribution is similar. The beginning part of Oops's curve is much lower than Apache and Squid which may indicate that Apache and Squid use more aggressive caching policies than Oops.

PT Proxy in fact is running the same code as our Multiplier and has no cache, thus generating a larger mean and median response time than Squid and Oops. The Collector will serve the responses with the same amount of delay as it collects the response from origin servers, and each time the pass-through proxy asks for an object from the Collector, it has to pay the same time as the Collector spends from the origin server, so, the PT Proxy and the Collector have similar distributions in the figure.

6 Related Work

There are several methods to evaluate Web proxies [10]. A number of researchers have proposed proxy cache (or more generally just Web) benchmark architectures [4, 3, 1, 2, 22, 6]. 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 correctly evaluating prefetching proxies.

Web Polygraph [26] 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 multiple Web cache-offs (e.g., [27]).

Koletsou and Voelker [18] 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.

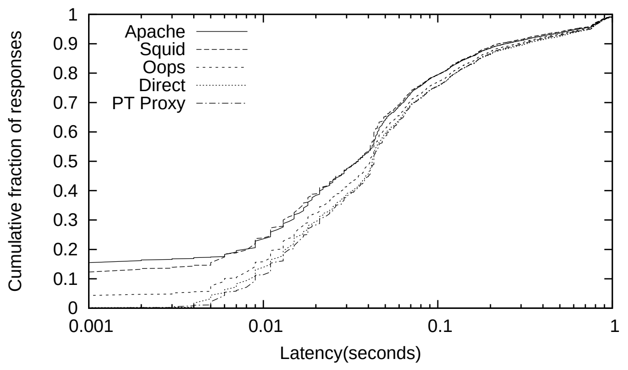


Figure 9: The cumulative distribution of response times for the tested proxies.

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 [19] also report on a personal proxy to measure client-perceived performance. Based on the Internet Junkbuster Proxy [16], 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.* [20] describe experiments measuring connect time and elapsed time for a number of workloads by replaying traces using Webjamma [15]. 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.

Patarin and Makpangou's Pandora platform [24] can measure the efficiency of proxy caches. A stack is located in front of the tested cache to catch the traffic between the clients and the cache; another one is located after the tested cache to get the traffic between the cache and origin servers. The measurements are based on the analysis of these two traffic traces. Compared to the SPE architecture, it can also test within a real environment with real requests. In addition, it can also test the performance of cooperating caches by analyzing the ICP traffic among these caches. However, their system does not monitor DNS resolution time which may be a problem when the Pandora system sends standard requests to the same origin servers repeatedly. More importantly, their system is unable to test multiple proxies simultaneously – instead, several separate tests are required (with likely different request loads and network conditions).

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 LEAP Multiplier can certainly be used for measuring client-side response times if placed adjacent to clients, it has had a slightly different target: the comparative performance evaluation of proxy caches.

LEAP is not the first implementation of the SPE architecture. An incomplete version was described with the SPE architecture design [9], and a more complete prototype called ROPE was described and tested in 2001 [12]. Compared to these attempts, LEAP has better performance and accuracy. For example, the architecture overhead is reported to be about 35ms in the ROPE implementation, while LEAP has an overhead around 15ms. We use a thread pool in our version of the Multiplier which can save time for creating threads while this time is unavoidable to the user of ROPE. LEAP can handle chunked-encoded responses from tested proxies, while ROPE cannot. LEAP will consider the DNS resolution time separately when calculating the response time, which will generate a more accurate timing than in ROPE. The “cache everything” function is more robust in the LEAP due to caching several complex responses from origin server, e.g., 404 responses. In addition to the thread pool, we implemented an event scheduling mechanism in LEAP’s Multiplier, while no such mechanisms existed in ROPE.

7 Discussion

In Section 2.1, we described our multi-threaded Multiplier, and later in Section 3.6 we explained why we used a thread pool. However, even with a pool of waiting threads, it is possible to have a workload that keeps many threads busy simultaneously, and so activate all threads in the pool. If this were to happen, subsequent requests would have to wait until an active thread completed its work. We pre-allocate 200 threads, with the expectation that our test workloads would not reach that level of usage. In practice, at 10 requests per second, httperf does find some connections refused by the current Multiplier. Additional experiments at lower request rates allow more connections to succeed, with similar experimental results to those presented.

The SPE architecture has the drawback of not being able to measure high workloads. This is because, by design, the Multiplier and Collector have to manage n+1 times the connections and bandwidth of the offered workload. Thus, the Multiplier and the Collector limit the request rate/bandwidth demand that can be used to test the proxies. Other mechanisms have been designed to find the failure point of web systems; LEAP measures performance when the systems are functional and not at their limits.

We have found that some of our measurement statistics are affected by error cases. That is, when timeouts occur, e.g., for DNS lookups, or connection attempts, or requests to an overloaded web server, such response times are much higher than the norm of a successful response. We include the high costs of error cases like these, although we recognize that they can sway measurements like the mean if the workload is too short. For that reason, we also tend to examine at least the median, and ideally plot the distribution of response times when making performance comparisons.

8 Summary

We have presented LEAP, an implementation of the SPE architecture. By design, LEAP is capable of testing multiple proxies simultaneously on the same workload using a live network connection. We have demonstrated the use of LEAP, finding that the Oops and Squid proxy caches have comparable performance. We have also described many of the challenges of developing a complex system and the methods used to resolve them in LEAP.

Along the way, we (re-)discovered a number of useful lessons, including:

- DNS delays are potentially significant, so they cannot be ignored.
- Don’t assume that an event-driven server will serve pending requests in any particular order.
- Be sure to consider all possible combinations of inputs.
- Sometimes it is better to control a variable than to model it.
- The good intentions of developers (to make their server robust) can sometimes get in the way.

- For best performance, pre-allocate and/or pre-execute code so that it is performed in advance of a user request.
- Incremental processing is often more efficient.
- In timing-sensitive applications, don't try to do everything all at once.
- Beware of syntactically different formats of semantically equivalent data.

While many of these are obvious, it is our hope that they will serve as useful reminders for future projects.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Apache Group. Apache HTTP server documentation. Online at `http://httpd.apache.org/docs/`, 2004.

[6] 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.

[7] T. Berners-Lee, R. T. Fielding, and H. Frystyk Nielson. Hypertext Transfer Protocol —HTTP/1.0. RFC 1945, May 1996.

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

[9] B. D. Davison. Simultaneous proxy evaluation. In *Proc. of the Fourth Int'l Web Caching Workshop (WCW99)*, pages 170–178, San Diego, CA, Mar. 1999.

[10] B. D. Davison. A survey of proxy cache evaluation techniques. In *Proc. of the Fourth Int'l Web Caching Workshop (WCW99)*, pages 67–77, San Diego, CA, Mar. 1999.

[11] B. D. Davison. Assertion: Prefetching with GET is not good. In A. Bestavros and M. Rabinovich, editors, *Web Caching and Content Delivery: Proc. of the Sixth Int'l Web Content Caching and Distribution Workshop (WCW'01)*, pages 203–215, Boston, MA, June 2001. Elsevier.

[12] B. D. Davison and C. Krishnan. ROPE: The Rutgers Online Proxy Evaluator. Technical Report DCS-TR-445, Dept. of Computer Science, Rutgers Univ., 2001.

[13] B. D. Davison, C. Krishnan, and B. Wu. When does a hit = a miss? In *Proc. of the Seventh Int'l Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, Aug. 2002.

[14] 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, June 1999.

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

[16] Junkbusters Corporation. The Internet Junkbuster proxy. Available from `http://www.junkbuster.com/ijb.html`, 2002.

[17] I. Khasilev. Oops proxy server homesite. Available from `http://www.oops-cache.org/`, 2002.

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

[19] R. Liston and E. Zegura. Using a proxy to measure client-side Web performance. In *Web Caching and Content Delivery: Proc. of the Sixth Int'l Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[20] B. Lui, G. Abdulla, T. Johnson, and E. A. Fox. Web response time and proxy caching. In *Proceedings of WebNet98*, Orlando, FL, Nov. 1998.

[21] C. Maltzahn and K. J. Richardson. Performance issues of enterprise level Web proxies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, Seattle, WA, June 1997. ACM Press.

[22] S. Manley, M. Seltzer, and M. Courage. A self-scaling and self-confi guring 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.

[23] D. Mosberger and T. Jin. httperf—A tool for measuring Web server performance. *Performance Evaluation Review*, 26(3):31–37, Dec. 1998.

[24] S. Patarin and M. Makpangou. On-line measurement of web proxy cache effi ciency. Research Report RR-4782, INRIA, Mar. 2003.

[25] R. Rivest. The MD5 message-digest algorithm. RFC 1321, Apr. 1992.

[26] A. Rousskov. Web Polygraph: Proxy performance benchmark. Online at `http://www.web-polygraph.org/`, 2004.

[27] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. Raw data and independent analysis at `http://www.measurement-factory.com/results/`, Dec. 2001.

[28] Y. Sato. DeleGate home page. Online at `http://www.delegate.org/`, 2004.

[29] D. Wessels. Squid Web proxy cache. Online at `http://www.squid-cache.org/`, 2004.