# Chapter 11

# SPE Implementation and Validation

## 11.1 Introduction

There are many possible methods to evaluate caching algorithms and implemented proxy caches, as we describe in Chapter 7. Of those that examine real systems, the typical approach is to apply benchmark tests (e.g. [AC98b, AC98a, BC98a, MJ98, GCR98, BD99, Rou02]) 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 [Rou02] workload (e.g., in a caching competition [RW00, RWW01]). Unfortunately, these approaches are unusable for evaluating proxy caches that preload content (such as [CY97, Cac02a]), and do not necessarily test on real-world workloads.

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 in Chapter 10 an approach using simultaneous proxy evaluation (SPE), potentially using a live network and client workload. In this chapter we describe an initial implementation of SPE, and our experiences with it. In the next sections we describe our application-layer implementation, and discuss issues that arose from the implementation and how we resolved them. We provide some baseline experiments in Section 11.4 to help validate its performance. In Section 11.5,

we demonstrate the use of our implementation 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.

## 11.2 Implementation

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 our system the Multiplier is a stand-alone multi-threaded program which is compatible with HTTP/1.1 [FGM+99] and HTTP/1.0 [BLFF96] protocols and was developed from scratch. The publicly available Squid proxy server [Wes02] version 2.5PRE7 was modified to develop the Collector.

### 11.2.1 The Multiplier

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 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 client-side persistent connections.

Figure 11.1 shows the headers of a sample request and response. The Multiplier

**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 11.1: Sample HTTP/1.0 request from Netscape and response via a Squid proxy.

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 [Riv92] 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 11.2 shows some sample lines from the Multiplier log.

### 11.2.2 The Collector

In this work, we have made changes only to the source code of Squid to implement our partial SPE tool. We have not modified the underlying operating system.

```
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 11.2: 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 differs).

Recall from Chapter 10 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 uncacheable responses.* This prevents multiple copies of uncacheable 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 connection setup time, response time and transfer time from the server for this object. In subsequent requests from clients for the same object in which the object is in the cache, we delay some time before sending the first chunk of the cached object, and we also delay some time before sending subsequent chunks, so that even though this request is a cache hit, the total response time experienced is the same as a cache miss.

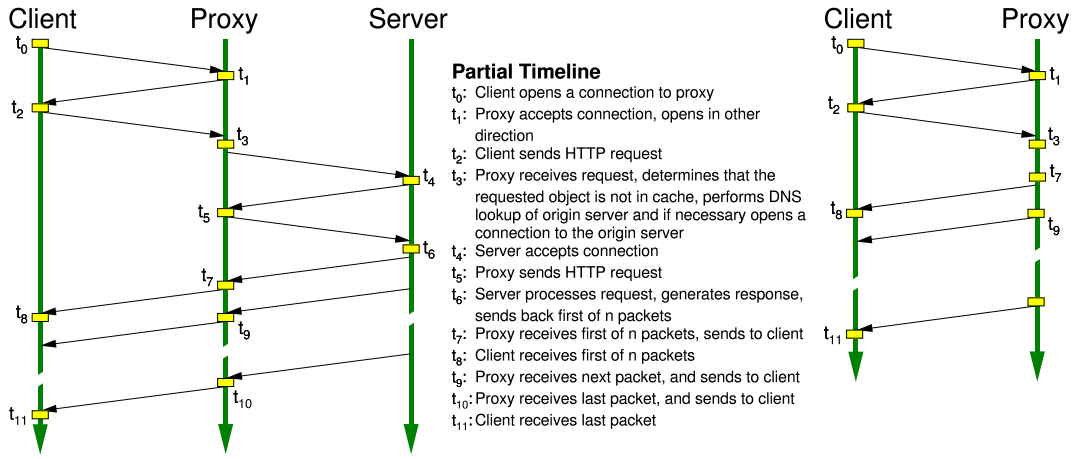
The function *clientSendMoreData* in *client_side.c* takes care of sending a chunk of data to the client. We have modified it so that for cached objects, the appropriate delays are introduced by using Squid's built-in event scheduling apparatus. Instead of sending the chunk right away, we schedule for the desired time the execution of a new function to send the chunk instead.

```
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 11.3: 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.

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 normally be 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 (a minimum *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).

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.




(a) Sequence for proxy cache miss.

(b) Sequence for proxy cache hit.

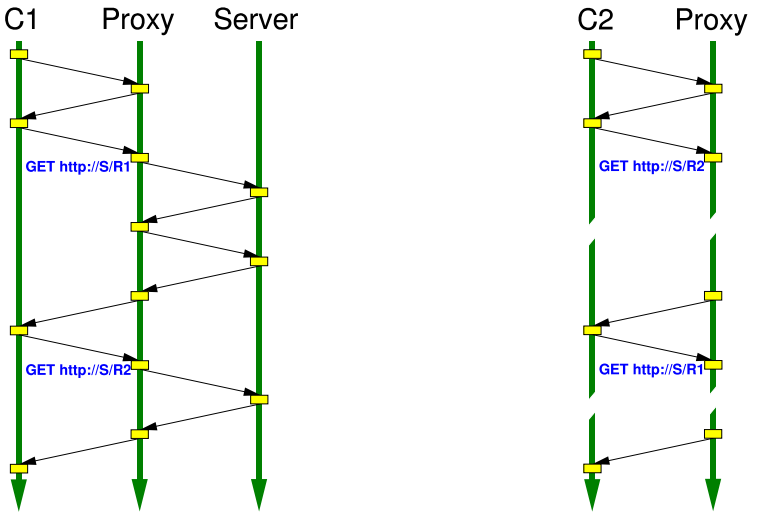
Figure 11.4: Transaction timelines showing the sequence of events to satisfy a client need. The diagrams do not show TCP acknowledgments, nor the packet exchange to close the TCP connection.

Similarly, requests with different cookies are sent to the origin server, even though the URLs are the same. Figure 11.3 shows some lines from a sample Collector log (identical to a standard Squid log).

## 11.3 Issues and Implementation

In order to reproduce miss timings as hits, we need to be careful about some of the details. In Figure 11.4a, we show a transaction timeline depicting the sequence of events when using a non-caching proxy (or equivalently, when the proxy does not have a valid copy of the content requested). In contrast, the transaction timeline in Figure 11.4b illustrates the typical sequence of events when a caching proxy has a valid copy of the requested content and returns it to the client.

In the case that the client has an idle persistent connection to the proxy, the transaction would start with $t_2$. If the proxy had an idle connection to the desired origin server, a new connection would not be necessary (merging the actions at $t_3$ and $t_5$ and eliminating the activity between). From the client's perspective, the time between $t_2$ and $t_0$ constitutes the *connection setup time*. The time between $t_8$ and $t_2$ constitutes the *initial response time*. The time between $t_{11}$ and $t_8$ is the *transfer time*. The complete




(a) Client re-uses persistent connection.

(b) Second client finds cached data.

Figure 11.5: Transaction timelines showing how persistent connections complicate timing replication.

response time would be the time elapsed between $t_{11}$ and $t_0$.

In the rest of this section, we discuss a number of the most significant issues encountered during implementation and how they have been addressed.

### 11.3.1 Persistent connections

**Issue:** Most Web clients, proxies and servers support persistent connections (which are optional in 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 establishment 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* [CKZ99], 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 11.5a). 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 11.5b). 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 either end. 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 SPE 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.

### 11.3.2 Request pipelining

**Issue:** 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.

**Implementation:** In our SPE implementation, 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). The impact of this choice is that we do not know whether the tested proxies support pipelined requests, but also means that the Multiplier does not need to be responsible for regenerating failed requests.

### 11.3.3 DNS resolution

**Issue:** The first request to an origin server requires a DNS lookup to determine the

server's IP address. In some cases, this process can take a noticeable amount of time [CK00]. For subsequent HTTP requests to the same server, no lookup will be required, as the IP address will have been saved in cache. It is important that even though one proxy may have issued a request that encountered a delay because of DNS resolution times, requests by other proxies should still experience the same delay.

**Implementation:** In order to provide no advantage to slower proxies that make a request later than a faster one, we incorporate the DNS resolution actions as part of the server connection establishment time. This helps to more accurately replicate the delay seen by the first request.

### 11.3.4 System scheduling granularity

**Issue:** Most operating systems have a limit on the granularity of task scheduling that is possible. Unless it is modified within the OS, this limit will provide a lower-bound on the ability to match previously recorded timings.

**Implementation:** In Linux as well as other UNIX-like systems on the Intel architecture, the `select(2)` system call has granularity of 10ms. Thus, if we tell `select` to wait at most 15ms, and no other monitored events occur, it will likely return after approximately 20ms. Thus, in our initial implementation a large difference would accumulate between what we wanted to delay and the actual amount of time delayed. In order to solve this problem, we use the following method: given some granularity, we must decide to round up or down. That is, for our development platform, if we need to delay 15ms, do we tune it to 10ms or to 20ms? In our implementation we decide this probabilistically. We first extract the largest multiple of the system-specific granularity from our desired delay. The fraction remaining is then used as a probability to determine whether to delay an additional timeslice, and thus the end result is a stochastically chosen delay value that is at most one timeslice larger or smaller than our original need.

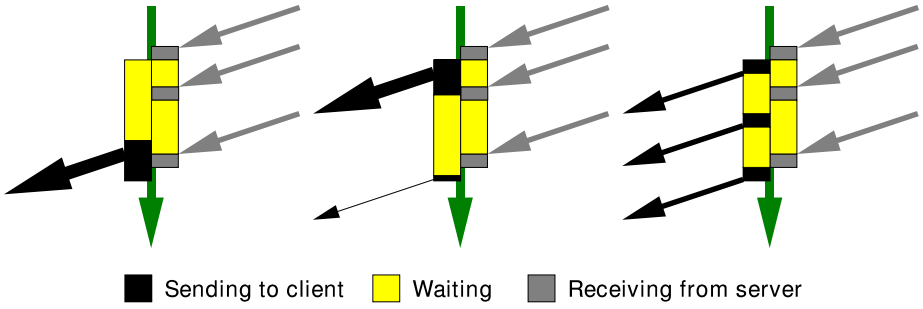
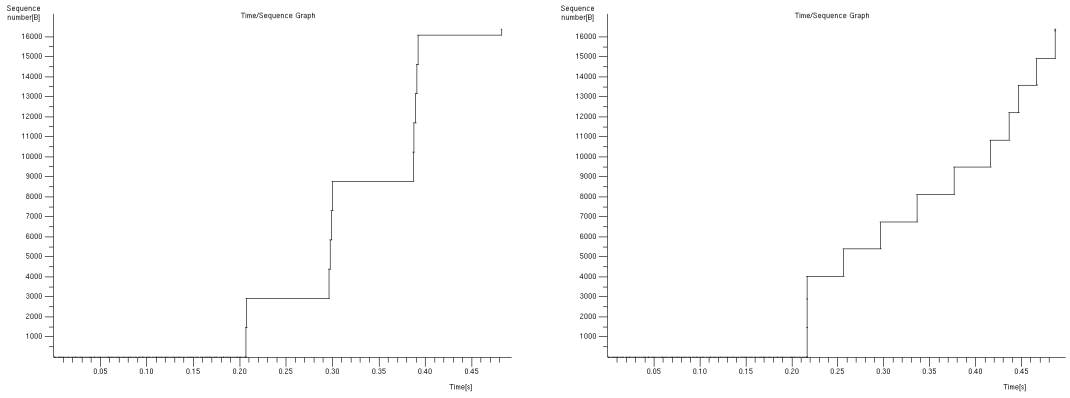



Figure 11.6: Possible inter-chunk delay schedules for sending to client.

### 11.3.5 Inter-chunk delay times

**Issue:** When Squid receives a non-trivial response from an origin server, it groups the data received in a single call into an object it calls a chunk, which it then stores into cache. Chunks then are the content of at minimum one packet, and possibly many packets, depending on the rate at which data is being received. In a cache miss, those chunks are sent on to the client as they are received. Since our goal is to replicate the miss experience as much as possible, it would be undesirable to wait and then send all data at the end, even though that might be a simplistic approach to replicate overall response time. Likewise, sending all data but the last byte until the desired time had passed would also be unreasonable, as it does not come close to replicating reality. Ideally, our system would send data at the same rate as it was received, and with the same inter-chunk delays. These three scenarios are depicted graphically in Figure 11.6. In each portion, the timings of when chunks were received are re-drawn on the right, while the various schedules for inter-chunk delays in the case of a cache miss are drawn on the left of the timeline.

The differences between these conditions are real — modern browsers will parse the HTML page as it is received (i.e., in chunks) to find embedded resources such as images to fetch, and when found, issue new requests for them. Prefetching proxies may in fact do something similar (e.g., [CY97, Cac02a]). Likewise, images with a progressive encoding may be rendered by a browser with low-resolution when the initial data is received and then be refined as the remaining data arrive.




(a) SPE Collector miss

(b) SPE Collector hit

Figure 11.7: Time/sequence graphs for one object.

**Implementation:** As described above, it may be insufficient to merely replicate the total response time — when data arrives within the response time period may be important. The overhead of recording the per-chunk delay was deemed too high for implementation. Instead, we replicate the average delay between chunks, as this only requires a fixed per-object storage overhead.

In order to accurately replicate the overall response time, we record the time difference between the time at which we receive the client request ($t_3$) and the time at which we begin to send the response ($t_7$) thus summing the server connection establishment time and the initial response time. In addition, we record the time difference between receiving the first chunk of response from the origin server ($t_7$) and the time when we receive the last chunk of response from the origin server ($t_{10}$) as the transfer time. As shown in Figure 11.4, this also corresponds to the time difference between just before sending the first chunk of response to the client and just before sending the last chunk of response to the client. So for subsequent requests for the same object, we intentionally add delays (in particular, before each chunk) so that the total time experienced between events $t_3$ and $t_{10}$ are identical to the miss case.

We insert delays in two situations: before the first chunk (i.e., before event $t_7$) we delay the amount of time it took for the connection establishment and initial response

time from the origin server. Before sending subsequent chunks, we delay a portion of the transfer time. In that situation the delay portion is calculated by the following formula:

$$delay = \frac{chunk\ size}{totalsize} * transfertime$$

In our experiments we found that the above formula is not accurate enough as there is a non-negligible chance that the randomly chosen errors (because of select call granularity) will accumulate. Therefore, we use a global optimization approach and change the formula to be:

$$delay = \frac{chunk\ size}{bytesremaining} * timeremaining$$

where *bytesremaining* is defined as the total remaining bytes of the cached object that has yet to be sent, and *timeremaining* is defined as the delay time remaining that is to be distributed to the remaining chunks of the object.

Figures 11.7a and b demonstrate the approach. In 11.7a, the Collector passes packets to the client as they are received from the origin, in effectively four blocks. In 11.7b, the Collector closely replicates the start and end of the transmission, but uses ten (smaller) blocks to do so.

### 11.3.6 Persistent state in an event-driven system

**Issue:** Our implementation requires the recording of various time values, some of which are object-based, and some of which are request-based. For example, we record which proxies have retrieved a particular object. Request-based information includes the *timeremaining* delay left to be applied to the current request.

**Implementation:** Since Squid uses a single process, we cannot easily use global storage for persistent state. Thus, we have extended the structures of `_connStateData`, `_StoreEntry`, and `_clientHttpRequest` in order to record the information for later access.

### 11.3.7 HTTP If-Modified-Since request

**Issue:** Not all HTTP requests return content. Instead, some respond only with header information along with response codes — telling the client that the object has moved (e.g., response code 302), or cannot be found (404), or that there is no newer version (304). The latter is of particular interest because it forces the consideration of many situations. 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 truth, 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. Likewise, the client may have a copy and need to verify its freshness with the proxy. If the proxy has a valid copy, it may wish to respond directly to the client, in which case the proxy needs to add delays as if the proxy had to confer with the origin server.

**Implementation:** We have considered IMS requests, and 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 no additional connection establishment and initial response time delay needs to be added. If the proxy is serving content that was cached, then as always, it needs to add transfer time delays (as experienced when the content was received). If the response served to the client does not contain content, then no transfer time is involved.

### 11.3.8 Network stability

**Issue:** An object may be cacheable for an arbitrary period of time. Since there exists significant variability in end-to-end network performance on the Internet [Pax99], network characteristics may have changed since the object was first retrieved.

**Implementation:** We use the times from the most recent origin server access of an object to be our gold standard. In reality, the origin server may be more or less busy and network routes to it may be better or worse. Thus, we have chosen consistent performance reproduction over technically accurate performance reproduction (which would preclude caching). Note that to minimize any adverse effects, we can limit the amount of time that we allow objects to be cached (since a new retrieval will get a new measurement of communication performance).

### 11.3.9 Caching of uncacheable responses

**Issue:** As described earlier, a SPE Collector must cache all responses, even those marked as uncacheable, to minimize the potential for duplicate requests to be sent to the origin server.

**Implementation:** In our Collector, the freshness time for uncacheable objects is set to a small amount of time (arbitrarily chosen to be 180 seconds) since we expect requests for such objects to appear from all interested proxies within a short time period and we don't really 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 (e.g., if one of the proxies prefetched an object that turned out to be uncacheable and the user 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.

### 11.3.10 Request scheduling

**Issue:** 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, a standard Squid proxy opens 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.

**Implementation:** Our Multiplier addresses this problem by sending the request only to the Collector first and only after receiving the complete response from the Collector, sends the request to each of the test proxies. Thus we ensure that when a test proxy passes a client request to the Collector, it will already be in the Collector's cache because the transfer of the object from the origin server is complete. Since the object is completely in cache, the Collector can serve subsequent requests from cache and avoid sending duplicate requests. However, this approach also implies that the inter-request timings seen by the tested proxies are not the same as those generated by the client, since the time that the proxy receives the request is dependent upon when the request is satisfied by the Collector.

### 11.3.11 Additional client latency

**Issue:** From Figure 10.1 which showed the SPE architecture and the communication paths between entities, 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.

**Implementation:** Though the SPE architecture introduces additional latency on the client's side, we expect it to be small since the SPE architecture typically runs in the same local network as the client. We will evaluate the overhead introduced by our implementation by running one instance within another in Section 11.4.4.

## 11.4 Validation

In this section, we describe our system configuration and three experiments that test and provide limited validation of our implementation.

### 11.4.1 Configuration and workloads

#### Software configuration

In the validation tests and in the experiments, we will use up to five proxy cache configurations. All are free, publicly available with source code. They are: Squid 2.4.STABLE6 [Wes02], Apache 1.3.23 (using mod_proxy) [Apa02], WcolD 961127_143211 [CY97], Oops 1.5.22 [Kha02], and DeleGate 7.9.9 [Sat02]. 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. WcolD is the only one of the five proxy caches to support link and embedded resource prefetching.

To generate workloads, we used two tools. The first is `httperf` [MJ98], which we use whenever generating an artificial workload. The httperf tool is highly configurable, supports HTTP/1.1 and persistent connections. In some experiments, we will collect the per-request timing results captured by httperf to calculate statistics on the overall response times as seen by the client.

The other is the `simclient` tool from the Proxicizer [Gad01, GCR98] toolset[1] since it is able to replay captured Web logs in close to real time, preserving the inter-request delays that were originally present.

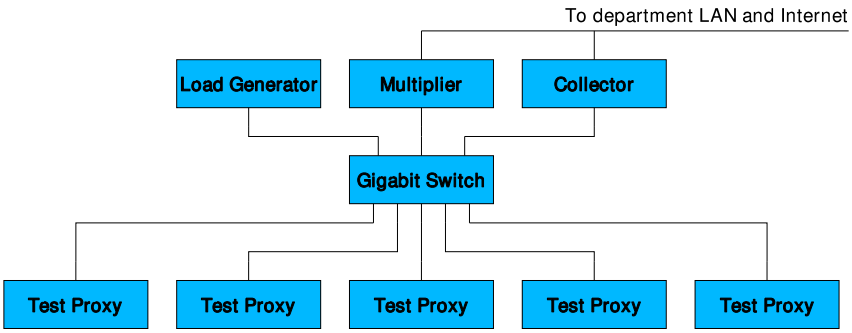



Figure 11.8: Experimental network configuration.

### Network configuration

For each test we run a Multiplier and a Collector as described above. As shown in Figure 11.8, each proxy runs on a separate system, as do each of the monitoring and load-generating processes. Each machine running a proxy under test has a single 1Ghz Pentium III with 512MB RAM and a single 40GB IDE drive. All machines are connected via full-duplex 100Mbit/sec Fast Ethernet to a gigabit switch using private IP space. A mostly stock RedHat 7.3 version of Linux was installed on each machine. The Multiplier and Collector ran on similar systems but with dual 1Ghz Pentium III processors, 2GB RAM, and gigabit Ethernet cards, and were dual-homed on the private network and on a LAN for external access.

### Data sets

In our experiments we use three representative data sets. To generate the first (entitled *Popular Queries*), we used the top fifty queries as reported by Lycos [Lyc02b] on 24 April 2002. Those queries were fed automatically to Google [Goo02], resulting in 10 URLs per query, for a total of 500 URLs that will be replayed by httperf. We eliminated three that caused problems for httperf in initial tests.

The second dataset (*IRCache*) was extracted from an NLANR IRCache proxy log (sv.ircache.net) on 18 April 2002. The intentions for this log of about half a million

[1]Proxycizer is a suite of applications and C++ classes that can be used in simulating and/or driving Web proxies.

entries were to focus on cacheable responses,[2] so we removed all log entries that had non-200 HTTP response codes or Squid status codes other than `TCP_HIT`, `TCP_MEM_HIT`, `TCP_IMS_HIT`, `TCP_REFRESH_HIT`, and `TCP_REFRESH_MISS`. This eliminated more than 75% of the log entries, leaving only those entries corresponding to objects that had been cached by the proxy or a client of the proxy. We then arbitrarily selected the URLs that ended in `.html` and eliminated any duplicate URLs. We additionally removed those that generated HTTP errors in initial tests, resulting in a set of 1054 URLs.

The third dataset (*Reverse proxy*) was selected to be a Web server workload, to be replayed in real time with simclient. We started with the access logs from the Apache [Apa02] Web server that runs a small departmental Web server, `www.cse.lehigh.edu`, for the month of May 2002. 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 94,000.

Since the simclient tool from the Proxycizer suite required logs in some proxy format, we converted the Apache logs to Squid logs. Unfortunately, Apache logs have 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 21-23) to be replayed through our SPE implementation. This represented just over 16,000 requests from 977 distinct client IP addresses.

A total of 196 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, as they represent a portion of what a real reverse proxy would encounter. While we did not analyze the effects that these robots had on the caching performance, Almeida *at al.* [AMR+01]

[2]Proxy logs from NLANR's IRCache proxies [Nat02] cannot be replayed in general because some entries are incomplete. To preserve privacy, all query terms (parts of the URL after a "?") are obscured.

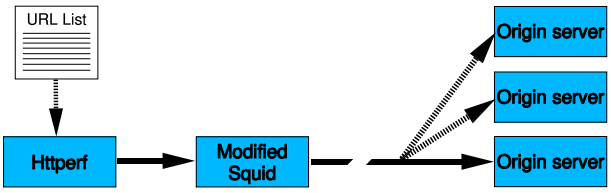



Figure 11.9: Experiment structure. httperf is used to send requests to our modified Squid and measure response times.

have found robot activity to have a significant impact.

### 11.4.2 Miss timing replication

In this section we test the ability of our Collector, a modified Squid proxy, to hide the use of its cache. To claim success, we must argue that proxies connecting to our Collector would be unlikely to tell whether the it utilizes a cache. To provide evidence supporting this argument, we experimentally measure response times of requests for data on real origin servers. For each run of our experiments, we iterate through a list of URLs (the IRCache workload) and give one at a time to httperf to send the request through our modified proxy (as shown in Figure 11.9.

We repeat the run for each of the four combinations of: original Squid versus modified Squid, and Squid with empty cache (i.e., generating all cache misses) versus Squid with cached results of previous run (making cache hits possible). Each run was repeated once per hour.

#### Results

After running the tests hourly for most of a day, we calculated relevant statistical properties using `strat` [MB01]. Examining first the IRCache data, we show the hit and miss response time distributions in Figure 11.10. Our primary concern, however, is in the paired differences in response times. That is, what is the typical difference in response time for a miss versus a replicated miss (i.e., a hit)? Since we are not concerned with whether the difference is positive or negative, we plot the distribution of the absolute value of the differences in Figure 11.11.

In Table 11.1 we show various summary statistics for the two datasets. Tests on

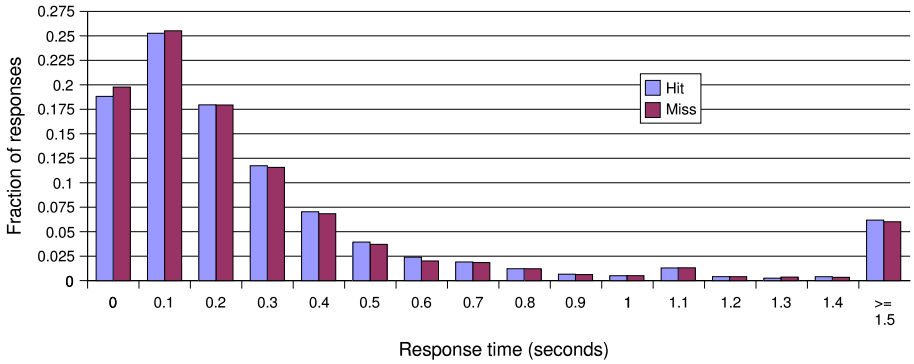



Figure 11.10: Distribution of cache hit and miss response times with NLANR IRCache data using our Collector.

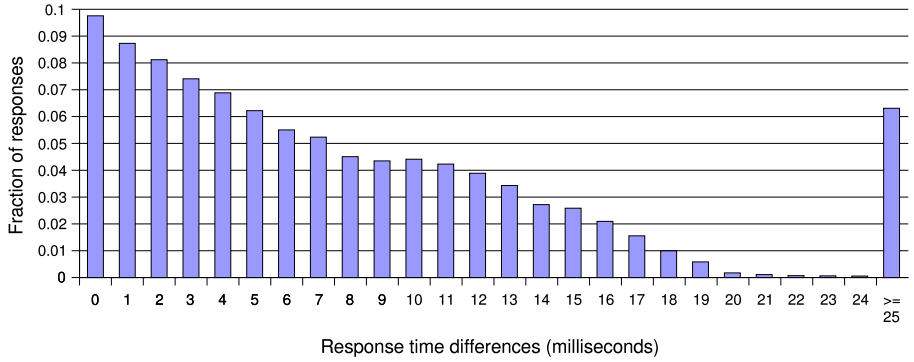



Figure 11.11: Distribution of absolute differences in paired response times between hits and misses with NLANR IRCache data using our Collector.

each dataset were repeated for some number of runs, and from each run we calculated the mean and median difference between response times of a cache miss versus a cache hit, and a cache miss versus a cache miss one hour later. In each row, we show the mean and standard error for the run means and the run medians. All values are in milliseconds.

| Data Set | # runs | Cache Miss vs. Hit (ms) Mean ± StdErr of run means | Cache Miss vs. Hit (ms) Mean ± StdErr of run medians | Cache Miss vs. Miss (ms) Mean ± StdErr of run means | Cache Miss vs. Miss (ms) Mean ± StdErr of run medians |
|---|---|---|---|---|---|
| Orig. IRCache | 13 | $12.85 \pm 4.41$ | $5.440 \pm 0.08$ | $-18.03 \pm 12.96$ | $0.08 \pm 0.16$ |
| Abs. Val. IRCache | 13 | $54.20 \pm 3.90$ | $6.45 \pm 0.07$ | $347.95 \pm 10.09$ | $10.53 \pm 0.04$ |
| Rev. Abs. IRCache | 13 | $7.44 \pm 0.82$ | $6.34 \pm 0.07$ | $337.84 \pm 11.23$ | $10.31 \pm 0.46$ |
| Popular Queries | 17 | $366.12 \pm 15.27$ | $13.01 \pm 0.19$ | $670.71 \pm 22.92$ | $33.34 \pm 1.39$ |

Table 11.1: Mean and standard error of run means and medians using our Collector.

| Data Set | # runs | Cache Miss vs. Hit (ms) Mean ± StdErr of run means | Cache Miss vs. Hit (ms) Mean ± StdErr of run medians | Cache Miss vs. Miss (ms) Mean ± StdErr of run means | Cache Miss vs. Miss (ms) Mean ± StdErr of run medians |
|---|---|---|---|---|---|
| Abs. Val. IRCache | 17 | $503.3 \pm 7.5$ | $245.6 \pm 1.0$ | $279.6 \pm 8.3$ | $10.3 \pm 0.5$ |
| Popular Queries | 17 | $600.2 \pm 17.6$ | $109.1 \pm 2.5$ | $578.1 \pm 17.2$ | $40.2 \pm 1.6$ |

Table 11.2: Mean and standard error of run means and medians using unmodified Squid.

For reference, the first row of Table 11.1 shows the results when calculating the real (not absolute value) difference between response times. The remaining rows calculate results using the absolute value of the differences. The data for the second row is otherwise identical, but now shows a significant difference in response times between the Miss-Hit and Miss-Miss cases. As expected, the typical difference for the Miss-Hit case is quite small (since our intention was to get this close to zero).

Recall from Section 11.4.1 that the IRCache data set was selected specifically to be cacheable. In fact, when we retrieved the URLs, in some instances the responses were not cacheable. If we remove these points, we get even better results, shown in the third row.

We now examine the data set generated from popular search engine queries. Unlike the IRCache data, this dataset contains many references (approximately two-thirds) to uncacheable objects. Under these conditions, we find the absolute difference in response times between miss and hit pairs of the same requests are much closer to the miss pairs separated by one hour, as shown in the last row.

For comparison, one might also ask what the typical difference is when an unmodified Squid is used (i.e., when hits are served as fast as possible). Results from testing on both datasets are recorded in Table 11.2. In both cases, mean differences in Miss-Hit response times are quite high (similar to Miss-Miss), and the medians are in fact much higher. This confirms the fact that unmodified hits are served much faster than misses, generating as much if not more variation in response times than misses an hour apart.

The above tests compared hit performance with that of a miss. It is also important to be able to generate multiple hits to the same object with consistent response times. The modified Squid is indeed able to achieve very similar response times for repeated

| Connection type | mean | median |
|---|---|---|
| dummy direct | 1057ms | 526ms |
| dummy proxy | 1108ms | 650ms |

Table 11.3: Comparison of response times between direct and proxy connections.

hits. The means of the run means and medians of the absolute difference between two hits for the same object is $6.60 \pm 0.64$ms and $2.03 \pm 0.29$, respectively.

From the above analysis, we can state that the typical difference in response times is significantly reduced for subsequent requests for the same object when our modified Squid Collector is utilized. In fact, when only tested on fully cacheable data, we find a useful mean miss-hit difference of just 7.4ms, and mean hit-hit difference of 6.6ms.

### 11.4.3 Proxy penalty

In this section we are concerned with the internal architecture of our SPE 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. We used the IRCache dataset and configured httperf to establish 1000 connections, with up to two requests per connection. Intervals between connection creation was randomly selected from a Poisson distribution with a mean delay of one second.

In this experiment we found that the direct connection was reported to be 50-125ms faster on average (shown in Table 11.3. From this result (of an admittedly small test), we might consider subtracting a factor of 50-100ms from the measured times of tested proxies, if we want to try to calculate exact timings. However, our primary concern is for comparative timings, which we address in Section 11.4.5.

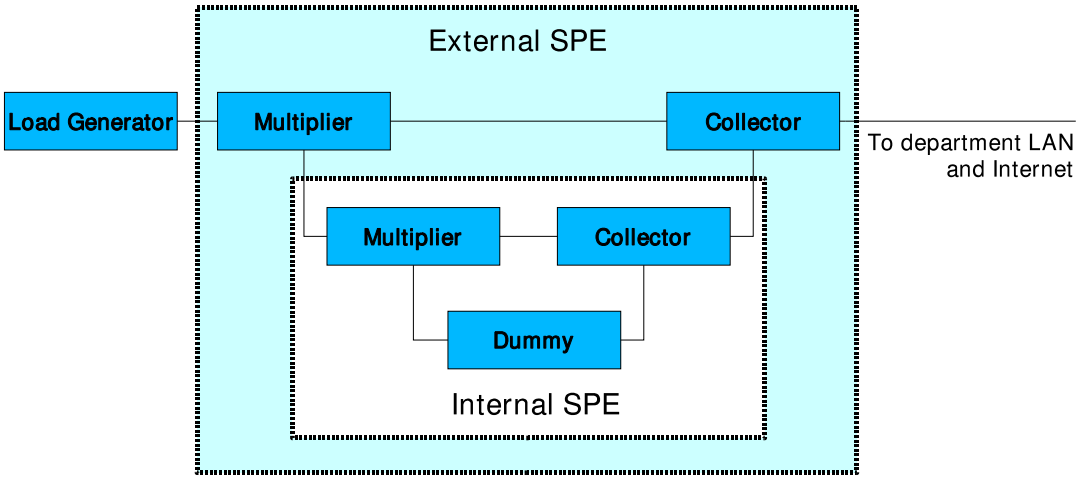



Figure 11.12: Configuration to test one SPE implementation within another.

### 11.4.4 Implementation overhead

In this experiment we are concerned with the general overhead that a SPE 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 SPE implementation using another SPE implementation, as shown in Figure 11.12. We again used the same IRCache artificial workload driven by httperf as input to the external SPE. The inner SPE (being measured) had just one Dummy proxy to drive. The outer one measured just the inner one.

The results of our test are presented in Table 11.4. We found that the inner SPE implementation generated a mean response time of 836ms, as compared to the direct connection response time of 721ms. Medians were smaller, at 444ms and 320ms, respectively. Thus, we estimate the overhead (for this configuration) to be approximately 120ms.

| Connection type | mean | median |
|---|---|---|
| direct | 721ms | 320ms |
| inner SPE | 836ms | 444ms |

Table 11.4: Evaluation of SPE implementation overhead.

| Connection type | mean | median |
|---|---|---|
| Apache proxy 1 | 1039ms | 703ms |
| Apache proxy 2 | 1040ms | 701ms |
| Apache proxy 3 | 1040ms | 701ms |
| Apache proxy 4 | 1038ms | 703ms |

Table 11.5: Performance comparison of identical proxies.

### 11.4.5 Proxy ordering effects

As described earlier, the Multiplier in our SPE 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 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 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 an artificial workload using httperf and the IRCache data to a set of identically configured Apache proxy caches.

In this experiment we found that all four proxies had very similar performance, as shown in Table 11.5. Mean response times were within two milliseconds, as were the median response times. Thus, we find that the results of identical systems with identical software are correctly reported to have essentially identical performance results.

## 11.5 Experiments

For the experiments using our SPE implementation to test proxies, we employed the same configurations as described in Section 11.4.1, except where noted.

| Proxy name | Bytes sent | Bytes received | Responses delivered | Mean time for response | Median time for response |
|---|---|---|---|---|---|
| (direct) | 100.00% | 100.00% | 100.00% | 529ms | 285ms |
| WcolD | 99.38% | 58.18% | 99.78% | 360ms | 61ms |
| Oops | 98.95% | 66.02% | 99.73% | 401ms | 180ms |
| DeleGate | 98.75% | 58.97% | 99.40% | 1045ms | 1039ms |
| Squid | 98.96% | 58.87% | 99.73% | 360ms | 106ms |
| Apache | 98.99% | 59.02% | 99.73% | 374ms | 199ms |

Table 11.6: Artificial workload results.

### 11.5.1 Proxies driven by synthetic workload

Here we wish to test the performance of various proxies using the httperf workload generator. For this experiment we tested Squid, DeleGate, Apache, WcolD, and an Oops proxy.

As in the previous validation tests, we use httperf to generate an artificial workload using the IRCache dataset. Since this dataset does not represent any real sequences of requests, we have disabled WcolD's prefetching mechanisms.

The results of this experiment are shown in Table 11.6 and graphically displayed in Figure 11.13. The timings shown are unadjusted results. **Bytes sent** measures the bytes delivered by the proxy to the Multiplier, relative to those sent by the Collector. Similarly, **Bytes received** measures the bytes received by the proxy from the Collector, relative to the bytes received by the Multiplier from the Collector which has no cache. Thus, a smaller fraction represents a higher byte caching ratio.

WcolD performs best under all measures. It ties with Squid for best mean response time at 360ms, but easily bests Squid's median of 106ms with a median of just 61ms. We record similar results for Apache's caching ability, but its mean response time is slightly higher at 374ms, and median response time is much higher at 199ms. While Oops provides the worst caching ability of the group, and a slightly higher mean response time at 401ms, its median response time is slightly better than Apache at 180ms. Finally, DeleGate is seen to provide similar bandwidth reductions, but has strikingly higher mean and median response times at just over one second.

DeleGate also shows a significant discontinuity in the cumulative response time

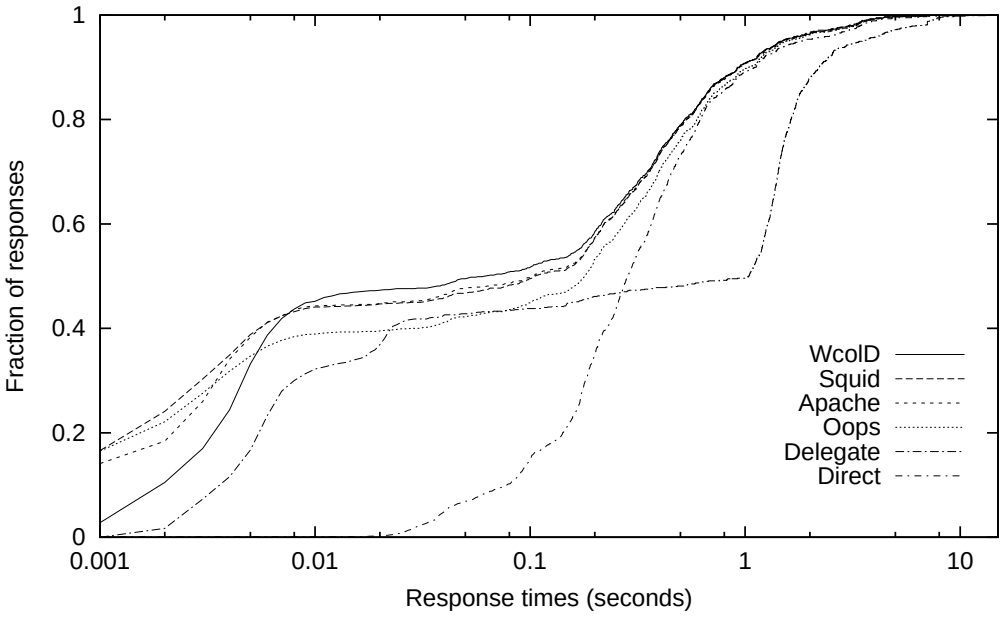



Figure 11.13: The cumulative distribution of response times for each proxy under the artificial workload.

distribution around 1 second in Figure 11.13. 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.

### 11.5.2 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 [Dav01a], we specifically chose to use a log from a Web server that predominantly generated static content.

The simclient tool was used to generate the workload, by replaying the requests in the Reverse proxy dataset. This allows us to preserve the inter-request delays (e.g., including user "thinking times") that were originally present, and matching more closely what a proxy would see in a real-world situation. In order to increase the load presented to the tested proxies, we will use three simclients simultaneously, each replaying requests from the first fifteen hours of a different day. We tested all five proxies, but in this case

permitted WcolD to prefetch embedded resources, but not links to new pages (since those pages could be on servers other than the one represented by this log).

Note that even though the requests in the workload included many HTTP/1.1 requests, the Proxycizer simclient only generates HTTP/1.0 requests. Additionally, unlike the original trace, the 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).

Likewise, some clients (such as Adobe Acrobat [Ado02]) 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, 2.3% of the responses were for partial content (HTTP response code 206).

The tests we have performed are relatively short and contain a small working set. As a result, the proxies do not significantly exercise their storage infrastructure or replacement policies. Thus, additional experiments are necessary before making significant conclusions about the proxies tested.

Instead, these tests have demonstrated some of the abilities of our SPE implementation to measure the simultaneous performance of implemented proxies. Unfortunately, because of limitations in the types of workloads used, the sample experiments reported here were unable to demonstrate improved performance from the use of prefetching. Note that we replayed the request log of a LAN-attached Web 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. As experience with and confidence in our SPE implementation grows, we anticipate larger-scale tests with a live user load, rather than replaying a trace, with content retrieved from remote servers.

Results from this experiment can be found in Table 11.7 and are graphically displayed in Figure 11.14. Since this workload was originally generated by a Web server

| Proxy name | Bytes sent | Bytes received | Resp. delivered | % resp. match | % not match | Mean lat. | Median lat. |
|---|---|---|---|---|---|---|---|
| (direct) | 100.00% | 100.00% | 100.00% | 100.00% | 0.00% | 433ms | 235ms |
| WcolD | 94.73% | 25.78% | 99.61% | 99.91% | 0.09% | 140ms | 9ms |
| Oops | 89.74% | 30.01% | 99.52% | 99.76% | 0.24% | 151ms | 9ms |
| DeleGate | 95.47% | 15.78% | 99.75% | 100.00% | 0.00% | 292ms | 16ms |
| Squid | 89.79% | 14.67% | 99.61% | 100.00% | 0.00% | 92ms | 7ms |
| Apache | 89.44% | 15.43% | 99.51% | 100.00% | 0.00% | 73ms | 7ms |

Table 11.7: Reverse proxy workload results.

with relatively few objects that were statically generated, it is highly cacheable, and so all of the caches perform significantly better than direct retrieval. Thus we see a significant benefit in placing any of these proxies in front of the Web server.

Squid does very well in this test, using the least external bandwidth, matching all responses to that sent by the Collector, and achieving the best median response time of 7ms, and an excellent mean response time of 92ms. However, Apache matches the median response time, and shortens Squid's mean response time by 19ms. This occurs in spite of higher external bandwidth usage. Examination of the experiment logs show

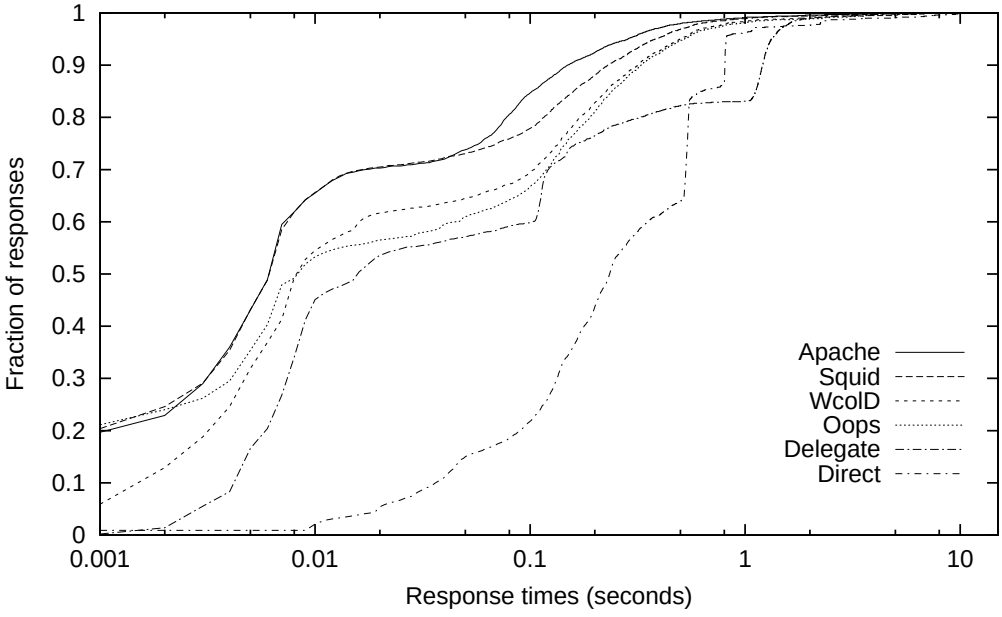



Figure 11.14: The cumulative distribution of response times for each proxy under the full-content reverse proxy workload.

an explanation. For requests with a URL containing a tilde (~), Apache apparently encodes the symbol as `%7E` when passing it on to the Collector. While equivalent from a Web server's point of view, Squid (and thus our Collector) does not recognize the equivalence of the two, and serves it as a real cache miss. Since this cache miss typically follows shortly after completion of the direct request, the response time from the Web server for this object is often better (since some or all of the relevant data may still be in the Web server's memory). The effect can also be seen in Figure 11.14 where Apache and Squid perform almost equivalently for more than 70% of the responses. The slower responses correspond to proxy cache misses, and Apache is serving misses more quickly than Squid.

WcolD and Oops come close with median response times of 9ms, but mean response times are worse (140ms and 151ms, respectively) and not all responses end up matching those sent by the Collector (.09% and .24%, respectively). They also use significantly more bandwidth than any of the others. DeleGate, while caching a similar number of bytes as Apache, has a higher median response time of 16ms and the worst mean response time of 292ms.

## 11.6 Summary

This chapter has described our implementation of SPE, and many of the issues related to it. We have described validation tests that we have performed on our implementation, as well as demonstrated the use of the current system in two experiments to simultaneously evaluate the performance of five publicly available proxy caches.

The primary contribution of this chapter has been the implementation, description, and evaluation of a SPE implementation. We have shown that the Collector can return cached results as if they were not cached. We have shown that the Multiplier measures performance of equivalent proxies equivalently, and that the overhead of running our system is relatively small. While we believe that even tighter results may be possible (particularly when the implementation includes operating system modifications), this chapter has demonstrated a credible complete SPE implementation.