The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

A Hierarchical Internet Object Cache

Anawat Chankhunthod, Peter Danzig, and Chuck Neerdaels
University of Southern California
Michael F. Schwartz and Kurt J. Worrell
University of Colorado, Boulder

A Hierarchical Internet Object Cache

Anawat Chankhunthod
Peter B. Danzig
Chuck Neerdaels
Computer Science Department
University of Southern California

Michael F. Schwartz
Kurt J. Worrell
Computer Science Department
University of Colorado - Boulder

Abstract: This paper discusses the design and performance of a hierarchical proxy-cache designed to make Internet information systems scale better. The design was motivated by our earlier trace-driven simulation study of Internet traffic. We challenge the conventional wisdom that the benefits of hierarchical file caching do not merit the costs, and believe the issue merits reconsideration in the Internet environment.

The cache implementation supports a highly concurrent stream of requests. We present performance measurements that show that our cache outperforms other popular Internet cache implementations by an order of magnitude under concurrent load. These measurements indicate that hierarchy does not measurably increase access latency. Our software can also be configured as a Web-server accelerator; we present data that our *httpd-accelerator* is ten times faster than Netscape's Netsite and NCSA 1.4 servers.

Finally, we relate our experience fitting the cache into the increasingly complex and operational world of Internet information systems, including issues related to security, transparency to cache-unaware clients, and the role of file systems in support of ubiquitous wide-area information systems.

1 Introduction

Perhaps for expedience or because software developers perceive network bandwidth and connectivity as free commodities, Internet information services like FTP, Gopher, and WWW were designed without caching support in their core protocols. The consequence of this misperception now haunts popular WWW and FTP servers. For example, NCSA, the home of Mosaic, moved to a multi-node cluster of servers to meet demand. NASA's Jet Propulsion Laboratory wide-area network links were saturated by the demand for Shoemaker-Levy 9 comet images in July 1994, and Starwave corporation runs a five-node SPARC-center 1000 just to keep up with demand for college basketball scores. Beyond distributing load away from server "hot spots", caching can also save bandwidth, reduce latency, and protect the network from clients that erroneously loop and generate repeated requests [9].

This paper describes the design and performance of the Harvest [5] cache, which we designed to make Internet information services scale better. The cache implementation is optimized to support a highly concurrent stream of requests with minimal queuing for OS-level resources, using non-blocking I/O, application-level threading and virtual memory management, and a Domain Naming System (DNS) cache. Because of its high performance, the Harvest cache can also be paired with existing HTTP servers (httpd's) to increase document server throughput by an order of magnitude.

Individual caches can be interconnected hierarchically to mirror an internetwork's topology, implementing the design motivated by our earlier NSFNET trace-driven simulation study [10].

1.1 Hierarchical Web versus File System Caches

Our 1993 study of Internet traffic showed that hierarchical caching of FTP files could eliminate half of all file transfers over the Internet's wide-area network links. [10]. In contrast, the hierarchical caching studies of Blaze and Alonso [2] and Muntz and Honeyman [17] showed that hierarchical caches can, at best, achieve 20% hit rates and cut file server workload in half. We believe the different conclusions reached by our study and these two file system studies lay in the workloads traced.

Our study traced wide-area FTP traffic from a switch near the NSFNET backbone. In contrast, Blaze and Alonso [2] and Muntz and Honeyman [17] traced LAN workstation file system traffic. While workstation file systems share a large, relatively static collection of files, such as *gcc*, the Internet exhibits a high degree of read-only sharing among a rapidly evolving set of popular objects. Because LAN utility files rarely change over a five day period, both [17] and [2] studies found little value of hierarchical caching over flat file caches at each workstation: After the first reference to a shared file, the file stayed in the local cache indefinitely and the upper-level caches saw low hit rates.

In contrast to workstation file systems, FTP, WWW, and Gopher facilitate read-only sharing of autonomously owned and rapidly evolving object spaces. Hence, we found that over half of NSFNET FTP traffic is due to sharing of read-only objects [10] and, since Internet topology tends to be organized hierarchically, that hierarchical caching can yield a 50% hit rate and reduce server load dramatically. Claffy and Braun reported similar statistics for WWW traffic [7], which has displaced FTP traffic as the largest contributor to Internet traffic. . .

Second, the cost of a cache miss is much lower for Internet information systems than it is for traditional caching applications. Since a page fault can take 10^5 times longer to service than hitting RAM, the RAM hit rate must be 99.99% to keep the average access speed at twice the cost of a RAM hit. In contrast, the typical miss-to-hit cost ratio for Internet information systems is 10:1 [1], and hence a 50% hit ratio will suffice to keep the average cost at twice the hit cost.

Finally, Internet object caching addresses more than latency reduction. As noted above and in the file system papers, hierarchical caching moves load from server hot spots. Not mentioned in the file system papers, many commercial sites proxy *all* access to the Web and FTP space through proxy caches, out of concern for Internet security. Many Internet sites are forced to use hierarchical object caches.

The Harvest cache has been in use for 1.5 years by a growing collection of about 100 sites across the Internet, as both a proxy-cache and as an httpd-accelerator. Our experiences during this time highlight several important issues. First, cache policy choices are made more difficult because of the prevalence of information systems that provide neither a standard means of setting object Time-To-Live (TTL) values, nor a standard for specifying objects as non-cacheable. For example, it is popular to create WWW pages that modify their content each time they are retrieved, by returning the date or access count. Such objects should not be cached. Second, because it is used in a wide-area network environment (in which link capacity and congestion vary greatly), cache topology is important. Third, because the cache is used in an administratively decentralized environment, security and privacy are important. Fourth, the widespread use of location-dependent names (in the form of Uniform Resource Locators, or URLs) makes it difficult to distinguish duplicated or aliased objects. Finally, the large number of implementations of both clients and servers leads to errors that worsen cache behavior.

We discuss these issues in more depth below.

2 Design

This section describes our design to make the Harvest cache fast, efficient, portable, and transparent.

2.1 Cache Hierarchy

To reduce wide-area network bandwidth demand and to reduce the load on Internet information servers, caches resolve misses through other caches higher in a hierarchy, as illustrated in Figure 1. In addition to the parent-child relationships illustrated in this figure, the cache supports a notion of *siblings*. These are caches at the same level in the hierarchy, provided to distribute cache server load.

Each cache in the hierarchy independently decides whether to fetch the reference from the object's home site or from its parent or sibling caches, using a simple *resolution protocol* that works as follows.

If the URL contains any of a configurable list of substrings, then the object is fetched directly from the object's home, rather than through the cache hierarchy. This feature is used to force the cache to resolve non-cacheable ("cgi-bin") URLs and local URLs directly from the object's home. Similarly, if the URL's domain name matches a configurable list of substrings, then the object is resolved through the particular parent bound to that domain.

Otherwise, when a cache receives a request for a URL that misses, it performs a remote procedure call to all of its siblings and parents, checking if the URL hits any sibling or parent. The cache retrieves the object from the site with the lowest measured latency.

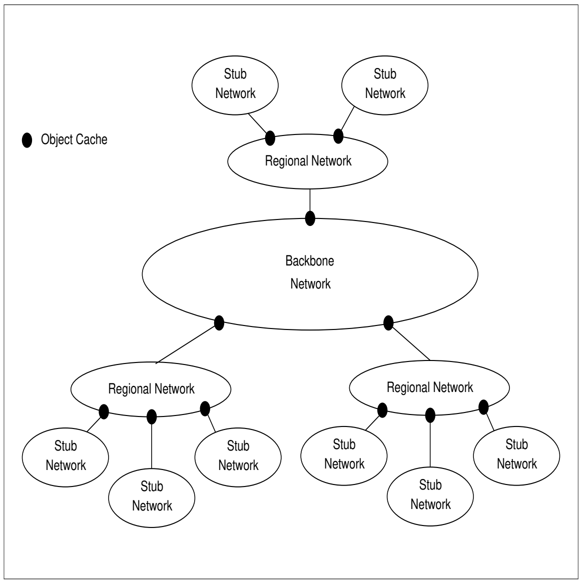


Figure 1: Hierarchical Cache Arrangement.

Additionally, a cache option can be enabled that tricks the referenced URL's home site into implementing the resolution protocol. When this option is enabled, the cache sends a "hit" message to the UDP echo port of the object's home machine. When the object's home echos this message, it looks to the cache like a hit, as would be generated by a remote cache that had the object. This option allows the cache to retrieve the object from the home site if it happens to be closer than any of the sibling or parent caches.

A cache resolves a reference through the first sibling, parent, or home site to return a UDP "Hit" packet or through the first parent to return a UDP "Miss" message if all caches miss and the home's UDP "Hit" packet fails to arrive within two seconds. However, the cache will not wait for a home machine to time out; it will begin transmitting as soon as all of the parent and sibling caches have responded. The resolution protocol's goal is for a cache to resolve an object through the source (cache or home) that can provide it most efficiently. This protocol is really a heuristic, as fast response to a ping indicates low latency. We plan to evolve to a metric that combines both response time and available bandwidth.

As will be shown in Section 3.5, hierarchies as deep as three caches add little noticeable access latency. The only case where the cache adds noticeable latency is when one of its parents fail, but the child cache has not yet detected it. In this case, references to this object are delayed by two seconds, the parent-to-child cache timeout. As the hierarchy deepens, the root caches become responsible for more and more clients. To keep root caches servers from becoming overloaded, we recommend that the hierarchy terminate at the first place in the regional or backbone network where bandwidth is plentiful.

2.2 Cache Access Protocols

The cache supports three access protocols: *encapsulating*, *connectionless*, and *proxy-http*. The *encapsulating* protocol encapsulates cache-to-cache data exchanges to permit

[1] This rough estimate is based on the observation that it takes about one second for a browser like Netscape to load an object from disk and render it for display, while a remote object takes about 10 seconds to be retrieved and displayed.

end-to-end error detection via checksums and, eventually, digital signatures. This protocol also enables a parent cache to transmit an object's remaining time-to-live to the child cache. The cache uses the UDP-based *connectionless* protocol to implement the parent-child resolution protocol. This protocol also permits caches to exchange small objects without establishing a TCP connection, for efficiency. While the *encapsulating* and *connectionless* protocols both support end-to-end reliability, the *proxy-http* protocol is the protocol supported by most Web browsers. In that arrangement, clients request objects via one of the standard information access protocols (FTP, Gopher, or HTTP) from a cache process. The term "proxy" arose because the mechanism was primarily designed to allow clients to interact with the WWW from behind a firewall gateway.

2.3 Cacheable Objects

The wide variety of Internet information systems leads to a number of cases where objects should not be cached. In the absence of a standard for specifying TTLs in objects themselves, the Harvest cache chooses not to cache a number of types of objects. For example, objects that are password protected are not cached. Rather, the cache acts as an application gateway and discards the retrieved object as soon as it has been delivered. The cache similarly discards URL's whose name implies the object is not cacheable (e.g. http://www.xyz.com/cgi-bin/query.cgi). for details about cacheable objects.). It is possible to limit the size of the largest cacheable object, so that a few large FTP objects do not purge ten thousand smaller objects from the cache.

2.4 Unique Object Naming

A URL does *not* name an object uniquely; the URL *plus* the MIME [2] header issued with the request uniquely identify an object. For example, a WWW server may return a text version of a postscript object if the client's browser is not able to view postscript. We believe that this capability is not used widely, and currently the cache does not insist that the request MIME headers match when a request hits the cache.

2.5 Negative Caching

To reduce the costs of repeated failures (e.g., from erroneously looping clients), we implemented two forms of negative caching. First, when a DNS lookup failure occurs, we cache the negative result for five minutes (chosen because transient Internet conditions are typically resolved this quickly). Second, when an object retrieval failure occurs, we cache the negative result for a parameterized period of time, with a default of five minutes.

2.6 Cache-Awareness

When we started designing the cache, we anticipated *cache-aware* clients that would decide between resolving an object indirectly through a parent cache or directly from the object's home. Towards this end, we created a version of Mosaic that could resolve objects through multiple caches, as illustrated in Figure 2. Within a few months, we reconsidered and dropped this idea as the number of new Web clients blossomed (cello, lynx, netscape, tkwww, etc.).

While no Web client is completely cache-aware, most support access through IP firewalls. Clients send all their

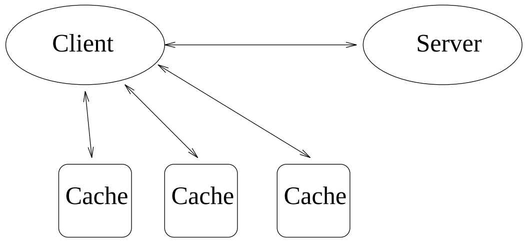


Figure 2: Cache-aware client

requests to their *proxy-server*, and the proxy-server decides how best to resolve it.

There are advantages and disadvantages to the cache-aware and cache-unaware approaches. Cache-unaware clients are simpler to configure; just set the proxy bindings that users already understand and which are needed to provide Web service through firewalls. On the other hand, cache-aware clients would permit load balancing, avoid the single point of failure caused by proxy caching, and (as noted in Section 2.2) allow a wider range of security and end-to-end error checking.

2.7 Security, Privacy, and Proxy-Caching

What is the effect of proxy-caching on Web security and privacy? WWW browsers support various authorization mechanisms, all encoded in MIME headers exchanged between browser and server. The *basic* authorization mechanism involves clear-text exchange of passwords. For protection from eavesdropping, the *Public Key* authorization mechanism is available. Here, the server announces its own public key in clear-text, but the rest of the exchange is encrypted for privacy. This mechanism is vulnerable to IP-spoofing, where a phony server can masquerade as the desired server, but the mechanism is otherwise invulnerable to eavesdroppers. Thirdly, for those who want both privacy and authentication, a *PGP* based mechanism is available, where public key exchange is done externally.

A *basic* authentication exchange follows the following dialog:

```
1. Client: GET <URL>

2. Server: HTTP:1.0 401 Unauthorized --
           Authentication failed

3. Client: GET <URL> Authorization:
           <7-bit-encoded name:password>

4. Server:  <returns a, b, c or d>
           a. Reply
           b. Unauthorized 401
           c. Forbidden 403
           d. Not Found 404
```

Given the above introduction to HTTP security mechanisms, we now explain how the cache transparently passes this protocol between browser and server.

When a server passes a `401 Unauthorized` message to a cache, the cache forwards it back to the client and purges the URL from the cache. The client browser, using the desired security model, prompts for a username and password, and reissues the GET URL with the authentication and authorization encoded in the request MIME header. The cache detects the authorization-related MIME header, treats it as

[2] MIME stands for "Multipurpose Internet Mail Extensions". It was originally developed for multimedia mail systems [4], but was later adopted by HTTP for passing typing and other meta data between clients and servers.

any other kind of non-cacheable object, returns the retrieved document to the client, but otherwise purges all records of the object. Note that under the clear-text *basic* authorization model, anyone, including the cache, could snoop the authorization data. Hence, the cache does not weaken this already weak model. Under the *Public Key* or *PGP* based models, neither the cache nor other eavesdroppers can interpret the authentication data.

Proxy-caching defeats IP address-based authentication, since the requests appear to come from the cache's IP address rather than the client's. However, since IP addresses can be spoofed, we consider this liability an asset of sorts. Proxy-caching does not prevent servers from encrypting or applying digital signature to their documents.

As a final issue, unless Web objects are digitally signed, an unscrupulous system administrator could insert invalid data into his proxy-cache. You have to trust the people who run your caches, just as you must trust the people who run your DNS servers, packet switches, and route servers. Hence, proxy-caching does not seriously weaken Web privacy.

2.8 Threading

For efficiency and portability across UNIX-like platforms, the cache implements its own non-blocking disk and network I/O abstractions directly atop a BSD *select* loop. The cache avoids forking except for misses to FTP URLs; we retrieve FTP URLs via an external process because the complexity of the protocol makes it difficult to fit into our select loop state machine. The cache implements its own DNS cache and, when the DNS cache misses, performs non-blocking DNS lookups (although without currently respecting DNS TTLs). As referenced bytes pour into the cache, these bytes are simultaneously forwarded to all sites that referenced the same object and are written to disk, using non-blocking I/O. The only way the cache will stall is if it takes a virtual memory page fault—and the cache avoids page faults by managing the size of its VM image (see Section 2.9). The cache employs non-preemptive, run-to-completion scheduling internally, so it has no need for file or data structure locking. However, to its clients, it appears multi-threaded.

2.9 Memory Management

The cache keeps all meta-data for cached objects (URL, TTL, reference counts, disk file reference, and various flags) in virtual memory. This consumes 48 bytes + strlen(URL) per object on machines with 32-bit words [3]. The cache will also keep exceptionally hot objects loaded in virtual memory, if this option is enabled. However, when the quantity of VM dedicated to hot object storage exceeds a parameterized high water mark, the cache discards hot objects by LRU until VM usage hits the low water mark. Note that these objects still reside on disk; just their VM image is reclaimed. The hot-object VM cache is particularly useful when the cache is deployed as an *httpd-accelerator* (discussed in Section 3.1).

The cache is write-through rather than write-back. Even objects in the hot-object VM cache appear on disk. We considered memory-mapping the files that represent objects, but could not apply this technique because it would lead to page-faults. Instead, objects are brought into cache via non-blocking I/O, despite the extra copies.

[3] We plan to replace the variable length URL with a fixed length MD5, reducing this number to 44+16=60 bytes/object.

Objects in the cache are referenced via a hash table keyed by URL. Cacheable objects remain cached until their cache-assigned TTL expires and they are evicted by the cache replacement policy, or the user manually evicts them by clicking the browser's "reload" button (the mechanism for which is discussed in Section 5.1). If a reference touches an expired Web object, the cache refreshes the object's TTL with an HTTP "get-if-modified".

The cache keeps the URL and per-object data structures in virtual memory but stores the object itself on disk. We made this decision on the grounds that memory should buy performance in a server-bottlenecked system: the meta-data for 1,000,000 objects will consume 60-80MB of real memory. If a site cannot afford the memory, then it should use a cache optimized for memory space rather than performance.

2.10 Disk Management

When disk space exceeds the high water mark, the cache enters its garbage collection mode. In this mode, every few dozen cache references, the cache discards the oldest objects encountered in a row of its object hash table. When disk usage drops below the low water mark, the cache exits from its garbage collection mode. If disk usage ever reaches the configured maxmimum, it immediately discards the oldest objects from the next row of the hash table.

The cache manages multiple disks and attempts to balance load across them. It creates 100 numbered directories on each disk and rotates object creation among the various disks and directories. Hence, an average directory of a cache that manages four disks and a million objects, stores 2500 numbered files. Since directory entries average about 24 bytes, an average directory may grow to 15, 4KB disk blocks. The number of files per directory can be increased to decrease directory size and reduce file system overhead.

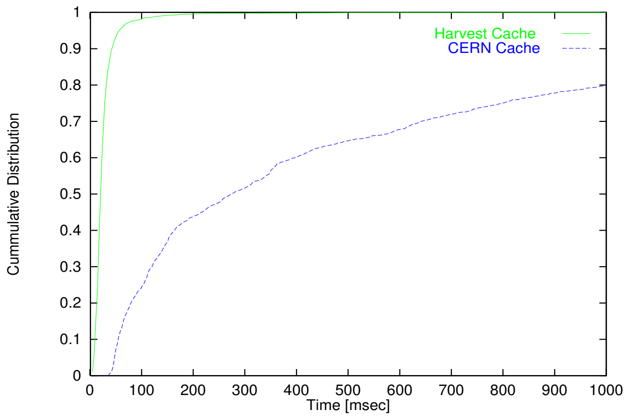
3 Performance

We now compare the performance of the Harvest cache against the CERN proxy-http cache [15] and evaluate the Harvest cache's *httpd-accelerator's* performance gain over the Netscape Netsite, NCSA 1.4, and CERN 3.0 Web servers. We also attempt to attribute the performance improvement to our various design decisions, as laid out in Section 2, and to evaluate the latency degradation of faulting objects through hierarchical caches. The measurements presented in this section were taken on SPARC 20 model 61 and SPARC 10 model 30 workstations. We will see that Harvest's high performance is mostly due to our disk directory structure, our choice to place meta-data in VM, and our threaded design.

3.1 Harvest vs. CERN Cache

To make our evaluation less dependent on a particular hit rate, we evaluate cache performance separately on hits and on misses for a given list of URLs. Sites that know their hit rate can use these measurements to evaluate the gain themselves. Alternatively, the reader can compute expected savings based on hit rates provided by earlier wide-area network traffic measurement studies [10, 7].

Figures 3 and 4 show the cumulative distribution of response times for hits and misses respectively. Figure 3 also reports both the median and average response times. Note that CERN's response time tail extends out to several seconds, so its average is three times its median. In the discussion below, we give CERN the benefit of the doubt and discuss median rather than average response times.



Measure	Harvest	CERN
Median	20 ms	280 ms
Average	27 ms	840 ms

Figure 3: Cumulative distribution of cache response times for hits, ten concurrent clients. The vertical-axis plots the fraction of events that take less than the time recorded on the horizontal-axis. For example, 60% of the time a CERN cache returns a hit in under 500 ms while 95% of the time a Harvest cache returns an object that hits in under 100 ms. CERN's long response time tail makes its average response time significantly larger than its median response time.

To compute Figure 4, ten clients concurrently referenced 200 unique objects of various sizes, types, and Internet locations against an initially empty cache. A total of 2,000 objects were faulted into the cache this way. Once the cache was warm, all ten clients concurrently referenced all 2,000 objects, in random order, to compute Figure 3. No cache hierarchy was used.

These figures show that the Harvest cache is an order of magnitude faster than the CERN cache on hits and on average about twice as fast on misses. We discuss the reasons for this performance difference in Section 3.4. We chose ten concurrent clients to represent a heavily accessed Internet server. For example, the JPL server holding the Shoemaker-Levy 9 comet images received a peak of approximately 4 requests per second, and the objects retrieved ranged from 50-150 kbytes.

For misses there is less difference between the Harvest and CERN caches because response time is dominated by remote retrieval costs. However, note the bump at the upper right corner of Figure 4. This bump comes about because approximately 3% of the objects we attempted to retrieve timed out (causing a response time of 75 seconds)—either due to unreachable remote DNS servers or unreachable remote object servers. While both the Harvest and the CERN caches will experience this long timeout the first time an object retrieval is requested, the Harvest cache's negative DNS and object caching mechanisms will avoid repeated timeouts issued within 5 minutes of the failed request. This can be important for caches high up in a hierarchy because long timeouts will tie up file descriptors and other limited system resources needed to serve the many concurrent clients.

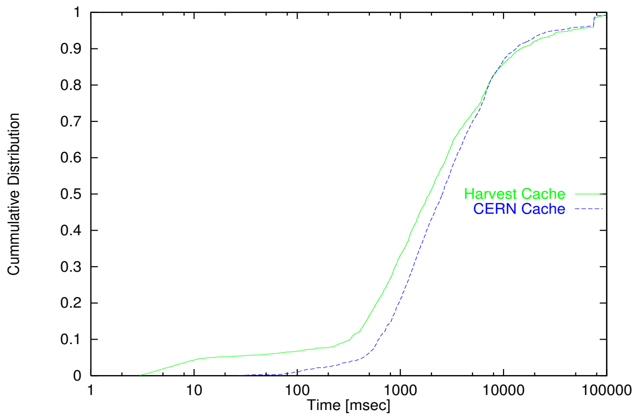


Figure 4: Cumulative distribution of cache response times for misses, ten concurrent clients, 2,000 URLs.

3.2 Httpd-Accelerator

This order of magnitude performance improvement on hits suggests that the Harvest cache can serve as an httpd *accelerator*. In this configuration the cache pretends to be a site's primary httpd server (on port 80), forwarding references that miss to the site's real httpd (on port 81). Further, the Web administrator renames all non-cacheable URLs to the httpd's port (81). References to cacheable objects, such as HTML pages and GIFs are served by the Harvest cache and references to non-cacheable objects, such as queries and cgi-bin programs, are served by the true httpd on port 81. If a site's workload is biased towards cacheable objects, this configuration can dramatically reduce the site's Web workload.

This approach makes sense for several reasons. First, by running the httpd-accelerator in front of the httpd, once an object is in the accelerator's cache all future hits will go to that cache, and misses will go to the native httpd. Second, while the httpd servers process forms and queries, the accelerator makes the simpler, common case fast [14]. Finally, while it may not be practical or feasible to change the myriad httpd implementations to use the more efficient techniques advocated in this paper, sites can easily deploy the accelerator along with their existing httpd.

While the benefit of running an httpd-accelerator depends on a site's specific workload of cacheable and non-cacheable objects, note that the httpd-accelerator cannot degrade a site's performance. Further note that objects that don't appear cacheable at first glance, can be cached at some slight loss of transparency. For example, given a demanding workload, accellerating access to "non-cacheable" objects like sport scores and stock quotes is possible if users can tolerate a short refresh delay, such as thirty seconds.

3.3 Httpd-Accelerator vs. Netsite & NCSA 1.4

Figure 5 demonstrates the performance of a Harvest cache configured as an httpd-accelerator. In this experiment, we faulted several thousand objects into a Harvest cache and then measured the response time of the Harvest cache versus the NCSA and Netsite httpd. Notice how Harvest serves documents that hit the httpd-accelerator with a median of 20 milliseconds, while the medians of Netscape's Netsite and NCSA's 1.4 httpd are each about 300 ms. On objects that miss, Harvest adds only about 20 ms to NCSA's and Netscape's 300 ms median access times.

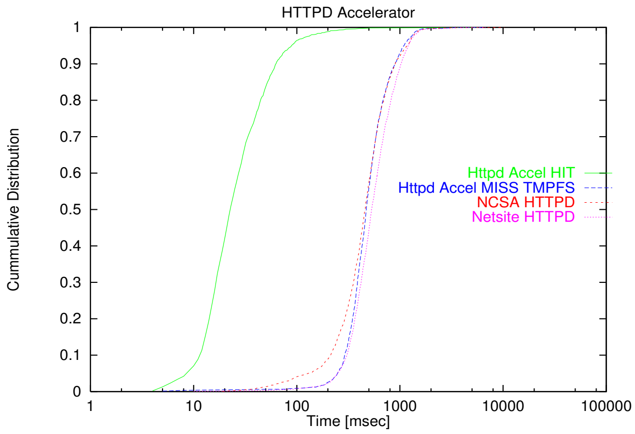


Figure 5: Response time of an httpd-accelerator versus a native httpd for a workload of all hits.

Note that on restart, the httpd-accelerator simply refaults objects from the companion Web server. For this reason, for added performance, the accellerator's disk store can be a memory based filesystem (TMPFS).

The httpd-accelerator can serve 200, small objects per second. We measured this by having 10 clients concurrently fetching two hundred different URL's, in random order, from a warm cache. Note that this is faster than implied by the average response time; we explain this below.

3.4 Decomposing Cache Performance

We now decompose how the Harvest cache's various design elements contribute to its performance. Our goal is to explain the roughly 260 ms difference in median and roughly 800 ms difference in average response times for the "all hits" experiment summarized in Figure 3.

The factor of three difference between CERN's median and average response time, apparent in CERN's long response time tail, occurs because under concurrent access, the CERN cache is operating right at the knee of its performance curve. Much of the response time above the median value corresponds to queueing delay for OS resources (e.g., disk accesses and CPU cycles). Hence, below, we explain the 260 ms difference between CERN's and Harvest's median response times (see Table 1).

Establishing and later tearing down the TCP connection between client and cache contributes a large part of the Harvest cache response time. Recall that TCP's three-way handshakes add a round trip transmission time to the beginning of a connection and a round trip time to the end. Since the Harvest cache can serve 200 small objects per second (5 ms per object) but the median response time as measured by cache clients is 20 ms, this means that 15 ms of the round-trip time is attributable to TCP connection management. This 15 ms is shared by both CERN and the Harvest cache.

We measured the savings of implementing our own threading by measuring the cost to `fork()` a UNIX process that opens a single file (`/bin/ls .`). We measured the savings from caching DNS lookups as the time to perform `gethostbyname()` DNS lookups of names pre-faulted into a DNS server on the local network. We computed the savings of keeping object meta-data in VM by counting the file system accesses of the CERN cache for retrieving meta-data from the UNIX file system. We computed the savings from caching hot objects in VM by measuring the file system accesses of the CERN cache to retrieve hot objects, excluding hits from the OS buffer pool.

Factor	Savings [msec.]
RAM Meta Data	112
Hot Object RAM Cache	112
Threading	36
DNS Lookup Cache	3
Total	264

Table 1: Back of the envelope breakdown of Performance Improvements

We first measured the number of file-system operations by driving cold-caches with a workload of 2,000 different objects. We then measured the number of file-system operations needed to retrieve these same 2,000 objects from the warm caches. The first, all-miss, workload measures the costs of writing objects through to disk; the all-hit workload measures the costs of accessing meta-data and objects. Because SunOS instruments NFS mounted file systems better than it instruments file systems directly mounted on a disk, we ran this experiment on an NFS-mounted file system. We found that the CERN cache averages 15 more file system operations per object for meta-data manipulations and 15 more file system operations per object for reading object data. Of course, we cannot convert operation counts to elapsed times because they depend on the size, state and write-back policy of the OS buffer pool and in-core inode table. (In particular, one can reduce actual disk I/O's by dedicating extra memory to file system buffering.) As a grotesquely coarse estimate, Table 1 assumes that disk operations average 15 ms and that half of the file system operations result in disk operations or 7.5 ms average cost per file system operation.

3.5 Cache Hierarchy vs. Latency

The benefits of hierarchical caching (namely, reduced network bandwidth consumption, reduced access latency, and improved resiliency) come at a price. Caches higher in the hierarchy must field the misses of their descendents. If the equilibrium hit rate of a leaf cache is 50%, this means that half of all leaf references get resolved through a second level cache rather than directly from the object's source. If the reference hits the higher level cache, so much the better, as long as the second and third level caches do not become a performance bottleneck. If the higher level caches become overloaded, then they could actually increase access latency, rather than reduce it.

Running on a dedicated SPARC 20, the Harvest cache can respond to over 250 UDP hit or miss queries per second, deliver as many as 200 small objects per second, and deliver 4 Mbits per second to clients. At today's regional network speeds of 1 Mbit/Second, the harvest cache can feed data to users four times faster than the regional network can get the data to the cache. Clearly, a Harvest cache is not a performance bottleneck. As an alternative way to look at the problem, in October 1995 the America Online network served 400,000 objects an hour during peak load. Depending on hit rate, a half dozen Harvest caches can support the entire AOL workload.

Figure 6 shows the response time distribution of faulting an object through zero, one and two levels of hierarchical caching. Figure 6 is read in two parts: access times from

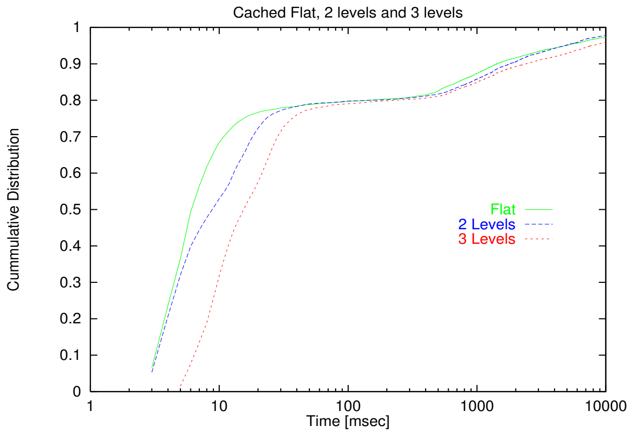


Figure 6: Effect of cache hierarchy on cache response time.

1-20 ms correspond to hits at a first level cache. Access times above 20 ms are due to hierarchical references that wind thier way through multiple caches and to the remote Web server. These measurements were gathered using five concurrent clients, each referencing the same 2,000 objects in a random order, against initially cold caches. The caches communicated across an Ethernet LAN, but the references were to distant objects. The result of this workload is that at least one client faults the object through multiple caches, but most of the clients see a first-level hit for that object.

Under this 80% hit workload, the average latency increases a few milliseconds per cache level (pick any point on the CDF axis and read horizontally across the graph until you cross the 1-level, 2-level, and 3-level lines.). For a 0% hit workload, each level adds 4-10 milliseconds of latency. Of course, if the upper-level caches are saturated or if the network connection to the upper-level cache is slow, these latencies will increase.

While this particular experiment does not correspond to any real workload, our point is that cache hierarchies do not significantly reduce cache performance on cache misses.

4 Cache Consistency

The Harvest cache, patterned after the Internet's Domain Naming System, employs TTL-based cache consistency. Like the Domain Naming system and the Alex File System, the Harvest cache can return stale data. If the object's MIME header conveys an expiration time, Harvest respects it. If the object's MIME header conveys its last modification time, Harvest sets the object's TTL to one half the time elapsed since the object was last modified. If the object carries neither time stamp, the cache assigns a default TTL according to the first regular expression that the URL matches. Unlike HTTP which can specify expiration times, Gopher and FTP provide no mechanism for owners to specify TTLs. Hence, when the Harvest cache fetches a Gopher or FTP object, it assigns it a default TTL based on regular expression. When the TTL expires, the cache marks the object as expired. For references to HTTP objects, the cache can revalidate expired objects using a "get-if-modifed" fetch. Get-if-modified fetches are propagated correctly through the hierarchy.

We believe that Web caches should not implement hierarchical invalidation schemes, but that sites exporting time-critical data should simply set appropriate expiration times. We say this because hierarchical invalidation, like AFS's call-back invalidations [13], requires state, and the Internet's basic unreliability and scale complicate stateful services. Below, we present data from two additional studies that support this position.

Object Lifetimes

In the absence of server-specified object expiration times, how should we set the cache's default TTL and how frequently should we "get-if-modified" versus simply returning the cached value? This question has several answers. Some service providers are contractually obligated to keep their cached data consistent: "We promise to get-if-modified if the cached value is over 15 minutes old". Others leave this question to the user and the brower's "reload" command. A quick study of Web data reveals that Internet object lifetimes vary widely and that no single mechanism will suffice.

To illustrate this, we periodically sampled the modification times of 4,600 HTTP objects distributed across 2,000 Internet sites during a three month period. We found that the mean lifetime of all objects was 44 days, HTML text objects and images were 75 and 107 days respectively, and objects of unknown types averaged 27 days. Over 28% of the objects were updated at least every 10 days, and 1% of the objects were dynamically updated. While WWW objects may become less volatile over time, the lifetime variance suggests that a single TTL applied to all objects will not be optimal. Note also that the time between modifications to particular objects is quite variable. This reinforces the need for picking cache TTLs based on last modification times, but that do not grow too large.

In the absence of object owner-provided TTLs, caches could grow object TTLs from some default minimum to some default maximum via binary exponential backoff. Each time an object's TTL expires, it could be revalidated with yet another get-if-modified.

Hierarchical Invalidation

In large distributed environments such as the Internet, systems designers have typically been willing to live with some degree of cache inconsistency to reduce server hot spots, network traffic, and retrieval latency. In reference [19], we evaluate hierarchical invalidation schemes for Internet caching, based on traces of Internet Web traffic and based on the topology of the current U.S. Internet backbone. In the absence of explicit expiration times and last-modification times, this study found that with a default cache TTL of 5 days, 20% of references are stale. Further, it found that the network cost of hierarchical invalidation exceeded the savings of Web caching for default cache TTLs shorter than five days.

Note that achieving a well-working hierarchical invalidation scheme will not be easy. First, hierarchical invalidation requires support from all data providers and caches. Second, invalidation of widely shared objects will cause bursts of synchronization traffic. Finally, hierarchical invalidation cannot prevent stale references, and would require considerable complexity to deal with machine failures.

At present we do not believe hierarchical invalidation can practically replace TTL based consistency in a wide-area distributed environment. However, part of our reluctance to recommend hierarchical invalidation stems from the current informal nature of Internet information. While most data available on the Internet today cause no problems even if stale copies are retrieved, as the Internet evolves to support more mission critical needs, it may make sense to try to overcome the hurdles of implementing a hybrid hierarchical invalidation mechanism for the applications that demand data coherence.

5 Experience

5.1 Transparency

Of our goals for speed, efficiency, portability and transparency, true transparency was the most difficult to achieve. Web clients expect caches and firewall gateways to translate FTP and Gopher documents into HTML and transfer them to the cache via HTTP, rather than simply forwarding referenced objects. This causes several problems. First, in HTTP transfers, a MIME header specifying an object's size should appear before the object. However, most FTP and Gopher servers cannot tell an object's size without actually transferring the object. This raises the following problem: should the cache read the entire object before it begins forwarding data so that it can get the MIME header right, or should it start forwarding data as soon as possible, possibly dropping the size-specifying MIME header? If the cache reads the entire object before forwarding it, then the cache may inject latency in the retrieval or, worse yet, the client may time out, terminate the transfer and lead the user to believe that the URL is unavailable. We decided not to support the size specification to avoid the timeout problem.

A related problem arises when an object exceeds the configured maximum cacheable object size. On fetching an object, once it exceeds the maximum object size, the cache releases the object's disk store but continues to forward the object to the waiting clients. This feature has the fortunate consequence of protecting the cache from Web applications that send endless streams only terminated by explicit user actions.

Web clients, when requesting a URL, transmit a MIME header that details the viewer's capabilities. These MIME headers differ between Mosaic and Netscape as well as from user to user. Variable MIME headers impact performance and transparency. As it happens, the Mosaic MIME headers range from a few hundred bytes to over a kilobyte and are frequently fragmented into two or more IP packets. Netscape MIME headers are much shorter and often fit in a single IP packet. These seemingly inconsequential details have a major impact that force us to trade transparency for performance.

First, if a user references an object first with Netscape and then re-references it with Mosaic, the MIME headers differ and officially, the cache should treat these as separate objects. Likewise, it is likely that two Mosaic users will, when naming the same URL, generate different MIME headers. This also means that even if the URL is a hit in a parent or sibling cache, correctness dictates that the requested MIME headers be compared. Essentially, correctness dictates that the cache hit rate be zero because any difference in any optional field of the MIME header (such as the user-agent) means that the cached objects are different because a URL does not name an object; rather, a URL plus its MIME header does. Hence, for correctness, the cache must save the URL, the object, *and* the MIME header. Testing complete MIME headers makes the parent-sibling UDP ping protocol expensive and almost wasteful. For these reasons, we do not compare MIME headers.

Second, some HTTP servers do not completely implement the HTTP protocol and close their connection to the client before reading the client's entire MIME header. Their underlying operating system issues a TCP-Reset control message that leads the cache to believe that the request failed. The longer the client's MIME header, the higher the probability that this occurs. This means that Mosiac MIME headers cause this problem more frequently than Netscape MIME headers. Perhaps for this reason, when it receives a TCP-Reset, Mosaic transparently re-issues the request with a short, Netscape-length MIME header. This leaves us with an unmaskable transparency failure since the cache cannot propagate TCP-Resets to its clients. Instead, the cache returns a warning message that the requested object may be truncated, due to a "non-conforming" HTTP server.

Third, current HTTP servers do not mark objects with a TTL, which would assist cache consistency. With the absence of help from the HTTP servers, the cache applies a set of rules to determine if the requested URL is likely a dynamically evaluated (and hence uncacheable) object. Some news services replace their objects many times in a single day, but their object's URLs do not imply that the object is not cacheable. When the user hits the client's "reload" button on Mosaic and Netscape, the client issues a request for the URL and adds a "don't-return-from-cache" MIME header that forces the cache to (hierarchically) fault in a fresh copy of an item. The use of the "reload" button is the least transparent aspect of the cache to users.

Fourth, both Mosaic and Netscape contain a small mistake in their proxy-Gopher implementations. For several months, we periodically re-reported the bug to Netscape Communications Corp., NCSA, and Spyglass, Inc., but none of these organizations chose to fix the bug. Eventually we modified the cache to avoid the client's bugs, forcing the cache to translate the Gopher and FTP protocols into properly formatted HTML.

Note that the Harvest cache's encapsulating protocol (see Section 2.2) supports some of the features that the proxy-http protocol sacrifices in the name of transparency. In the future, we may change cache-to-cache exchanges to use the encapsulating protocol.

5.2 Deployment

As Harvest caches get placed in production networks, we continue to learn subtle lessons. Our first such lesson was the interaction of DNS and our negative cache. When users reference distant objects, occasionally the cache's DNS resolver times out, reports a lookup failure, and the cache adds the IP address to its negative IP cache. However, a few seconds later, when the user queries DNS directly, the server returns a quick result. What happened, of course, is that the nameserver resolves the name between the time that the cache gives up and the time that the frustrated user queries the name server directly. Our negative cache would continue to report that the URL was unresolvable for the configured negative caching interval. Needless to say, we quickly and dramatically decreased the default negative cache interval.

At other times Harvest caches a DNS name that it managed to resolve through an inconsistent, secondary name server for some domain. When the frustrated user resolves the name directly, he may get his reply from a different, consistent secondary server. Again the user reports a bug that Harvest declares that a name is bad when it is not.

Caches serving intense workloads overflowed the operating system's open file table or exhausted the available TCP port number space. To solve these problems, we stopped accepting connections to new clients once the we approach the file descriptor or open file table limit. We also added watchdog timers to shut down clients or servers that refused to close their connections. As people port the cache to various flavors of UNIX, some had to struggle to get non-blocking disk I/O correctly specified.

Deploying any Web cache–Harvest, Netscape, or CERN–for a regional network or Internet Service Provider is tricky business. As Web providers customize their pages for particular browsers, maintaining a high cache hit rate will become harder and harder.

5.3 Open Systems vs. File Systems

The problems we faced in implementing the Harvest object cache were solved a decade ago in the operating systems community, in the realm of distributed file systems. So the question naturally arises, "Why not just use a file system and dump all of this Web silliness?" For example, Transarc proposes AFS as a replacement for HTTP [18].

AFS clearly provides better caching, replication, management, and security properties than the current Web does. Yet, it never reached the point of exponential growth that characterizes parts of the Internet infrastructure, as has been the case with TCP/IP, DNS, FTP, Gopher, WWW, and many other protocols and services. Why would the Internet community prefer to rediscover and re-implement all of the technologies that the operating systems community long ago solved?

Part of the answer is certainly that engineers like to reinvent the wheel, and that they are naturally lazy and build the simplest possible system to satisfy their immediate goals. But deeper than that, we believe the answer is that the Internet protocols and services that become widespread have two characterizing qualities: simplicity of installation/use, and openness. As a complex, proprietary piece of software, AFS fails both tests.

But we see a more basic, structural issue: We believe that file systems are the wrong abstraction for ubiquitous information systems. They bundle together a collection of features (consistency, caching, etc.) in a way that is overkill for some applications, and the only way to modify the feature set is either to change the code in the operating system, or to provide mechanisms that allow applications selective control over the features that are offered (e.g., using `ioctls` and kernel build-time options). The Internet community has chosen a more loosely coupled way to select features: a la carte construction from component technologies. Rather than using AFS for the global information service, Internet users chose from a wealth of session protocols (Gopher, HTTP, etc.), presentation-layer services (Kerberos, PGP, Lempel-Ziv compression, etc.), and separate cache and replication services. At present this has lead to some poor choices (e.g., running the Web without caching support), but economics will push the Internet into a better technical configuration in the not-too-distant future. Moreover, in a rapidly changing, competitive multi-vendor environment it is more realistic to combine features from component technologies than to wrap a "complete" set in an operating system.

6 Related Efforts

There has been a great deal of research into caching. We restrict our discussion here to wide-area network caching efforts.

One of the earliest efforts to support caching in a wide-area network environment was the Domain Naming System [16]. While not a general file or object cache, the DNS supports caching of name lookup results from server to server and also from client to server (although the widespread BIND *resolver* client library does not provide client caching), using timeouts for cache consistency.

AFS provides a wide-area file system environment, supporting whole file caching [13]. Unlike the Harvest cache, AFS handles cache consistency using a server callback scheme that exhibits scaling problems in an environment where objects can be globally popular. The Harvest cache implementation we currently make available uses timeouts for cache consistency, but we also experimented with a hierarchical invalidation scheme (see Section 4). Also, Harvest implements a more general caching interface, allowing objects to be cached using a variety of access protocols (FTP, Gopher, and HTTP), while AFS only caches using the single AFS access protocol.

Gwertzman and Seltzer investigated a mechanism called *geographical push caching* [12], in which the server chooses to replicate documents as a function of observed traffic patterns. That approach has the advantage that the choice of what to cache and where to place copies can be made using the server's global knowledge of reference behavior. In contrast, Bestavros et al. [11] explored the idea of letting clients make the choice about what to cache, based on application-level knowledge such as user profiles and locally configured descriptions of organizational boundaries. Their choice was motivated by their finding that cache performance could be improved by biasing the cache replacement policy in favor of more heavily shared local documents. Bestavros also explored a mechanism for distributing popular documents based on server knowledge [3].

There have also been a number of simulation studies of caching in large environments. Using trace-driven simulations Alonso and Blaze showed that server load could be reduced by 60-90% [1, 2]. Muntz and Honeyman showed that a caching hierarchy does not help for typical UNIX workloads [17]. A few years ago, we demonstrated that FTP access patterns exhibit significant sharing and calculated that as early as 1992, 30-50% of NSFNET traffic was caused by repeated access to read-only FTP objects [10].

There have also been several network object cache implementations, including the CERN cache [15], Lagoon [6], and the Netscape client cache. Netscape currently uses a 5 MB default disk cache at each client, which can improve client performance, but a single user might not have a high enough hit rate to affect network traffic substantially. Both the CERN cache and Lagoon effort improve client performance by allowing alternate access points for heavily popular objects. Compared to a client cache, this has the additional benefit of distributing traffic, but the approach (forking server) lacks required scalability. Harvest is unique among these systems in its support for a caching hierarchy, and in its high performance implementation. Its hierarchical approach distributes and reduces traffic, and the non-blocking/non-forking architecture provides greater scalability. It can be used to increase server performance, client performance, or both.

Cate's *Alex file system* [8], completed before the explosive growth of the Web, exports a cache of anonymous FTP space via an NFS interface. For performance, Alex caches IP addresses, keeps object meta-data in memory, and caches FTP connections to remote servers to stream fetches to multiple files. Alex uses TTL-based consistency, caching files for one tenth of the elapsed time between the file was fetched and the file's creation/modification date. The architecture of the Harvest cache is similar to Alex in many ways: Harvest caches IP addresses, keeps meta-data in memory, and implements a similar life-time based object consistency algorithm. Harvest does not stream connections to Gopher and Web servers, because these protocols do not yet support streaming access. In contrast to Alex, which exports FTP files via the UDP-based NFS protocol, Harvest exports Gopher, FTP, and Web objects via the proxy-http interface implemented by Web browsers. Furthermore, the Harvest cache supports hierarchical caching, implements a consistency protocol tailored for Web objects, and serves as a very fast httpd-accelerator.

7 Summary

Internet information systems have evolved so rapidly that they postponed performance and scalability for the sake of functionality and easy deployment. However, they cannot continue to meet exponentially growing demand without new infrastructure. Towards this end, we designed the Harvest hierarchical object cache.

This paper presents measurements that show that the Harvest cache achieves better than an order of magnitude performance improvement over other proxy caches. It also demonstrates that HTTP is *not* an inherently slow protocol, but rather that many popular implementations have ignored the sage advice to make the common case fast [14].

Hierarchical caching distributes load away from server hot spots raised by globally popular information objects, reduces access latency, and protects the network from erroneous clients. High performance is particularly important for higher levels in the cache hierarchy, which may experience heavy service request rates.

The Internet's autonomy and scale present difficult challenges to the way we design and build system software. Once software becomes accepted as de facto standards, both its merits and its deficiencies may live forever. For this reason, the real-world complexities of the Internet make one face difficult design decisions. The maze of protocols, independent software implementations, and well-known bugs that comprise the Internet's upper layers, frequently force tradeoffs between design cleanliness and operational transparency. This paper discusses many of the tradeoffs forced upon us.

Software and Measurement Data Availability

The Harvest cache runs under several operating systems, including SunOS, Solaris, DEC OSF-1, HP/UX, SGI, Linux, and IBM AIX. Binary and source distributions of the cache are available from http://excalibur.usc.edu/. The test code and the list of URLs employed in the performance evaluation presented here are available from http://excalibur.usc.edu/experiments. General information about the Harvest system, including the cache user's manual, is available from http://harvest.cs.colorado.edu/

Acknowledgments

This work was supported in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Danzig was also supported in part by the Air Force Office of Scientific Research under Award Number F49620-93-1-0082, and by a grant from Hughes Aircraft Company under NASA EOSDIS project subcontract number ECS-00009, and by National Science Foundation Institutional Infrastructure Grant Number CDA-921632. Schwartz was also supported in part by the National Science Foundation under grant numbers NCR-9105372 and NCR-9204853, an equipment grant from Sun Microsystems' Collaborative Research Program, and from the University of Colorado's Office of the Vice Chancellor for Academic Affairs. Chuck Neerdaels was supported by HBP NIH grant 1-P20-MH/DA52194-01A1.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

We thank John Noll for writing the initial cache prototype. We thank Darren Hardy, Erhyuan Tsai, and Duane Wessels for all of the work they have done on integrating the cache into the overall Harvest system.

References

[1] Rafael Alonso and Matthew Blaze. Long-term caching strategies for very large distributed file systems. *Proceedings of the USENIX Summer Conference*, pages 3–16, June 1991.

[2] Rafael Alonso and Matthew Blaze. Dynamic hierarchical caching for large-scale distributed file systems. *Proceedings of the Twelvth International Conference on Distributed Computing Systems*, June 1992.

[3] Azer Bestavros. *Demand-Based Document Dissemination for the World-Wide Web*. Computer Science Department, Boston University, February 1995. Available from ftp://cs-ftp.bu.edu/techreports/95-003-web-server-dissemination.ps.Z.

[4] Nathaniel Borenstein and Ned Freed. RFC 1521: MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies, September 1993.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994. Available from ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Conf.ps.Z.

[6] Paul M. E. De Bra and Reiner D. J. Post. *Information Retrieval in the World-Wide Web: Making client-based searching feasible*. Available from http://www.win.tue.nl/win/cs/is/reinpost/www94/www94.html.

[7] Hans-Werner Braun and Kimberly Claffy. Web traffic characterization: an assessment of the impact of caching documents from NCSA's web server. In *Second International World Wide Web Conference*, October 1994.

[8] Vincent Cate. Alex - a global filesystem. *Proceedings of the Usenix File Systems Workshop*, pages 1–11, May 1992.

[9] Peter B. Danzig, Katia Obraczka, and Anant Kumar. An analysis of wide-area name server traffic: A study of the Domain Name System. *ACM SIGCOMM 92 Conference*, pages 281–292, August 1992.

[10] Peter B. Danzig, Michael F. Schwartz, and Richard S. Hall. A case for caching file objects inside internetworks. *ACM SIGCOMM 93 Conference*, pages 239–248, September 1993.

[11] Bestavros et al. Application-level document caching in the Internet. *Workshop on Services in Distributed and Networked Environments, Summer 1995. Available from ftp://cs-ftp.bu.edu/techreports/95-002-web-client-caching.ps.Z*, January 1995.

[12] James Gwertzman and Margo Seltzer. The case for geographical push-caching. *HotOS Conference*, 1994. Available as ftp://das-ftp.harvard.edu/techreports/tr-34-94.ps.gz.

[13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[14] Bulter Lampson. Hints for computer system design. *Operating Systems Review*, 17(5):33–48, Oct 10-13, 1983.

[15] Ari Luotonen, Henrik Frystyk, and Tim Berners-Lee. CERN HTTPD public domain full-featured hypertext/proxy server with caching, 1994. Available from http://info.cern.ch/hypertext/WWW-/Daemon/Status.html.

[16] Paul Mockapetris. RFC 1035: Domain names - implementation and specification. Technical report, University of Southern California Information Sciences Institute, November 1987.

[17] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[18] Mirjana Spasojevic, Mic Bowman, and Alfred Spector. *Information Sharing Over a Wide-Area File System*. Transarc Corporation, July 1994. Available from ftp://grand.central.org/darpa/arpa2/papers-/usenix95.ps.

[19] Kurt Jeffery Worrell. *Invalidation in Large Scale Network Object Caches*. Department of Computer Science, University of Colorado, Boulder, Colorado, December 1994. M.S. Thesis, available from ftp://ftp.cs.colorado.edu/pub/cs/techreports-/schwartz/WorrellThesis.ps.Z.

Anawat Chankhunthod received his B.Eng in Electrical Engineering from the Chiang Mai University, Thailand in 1991 and his M.S in Computer Engineering from the University of Southern California in 1994. He is currently a Ph.D candidate in Computer Engineering at the University of Southern California. Shortly after receiving his B.Eng, he joined the faculty of the Department of Electical Engineering, Chiang Mai University and currently is on leave for extending his education. He is currently an research assistant at the Networking and Distributed system laboratory at the University of Southern California. His research focuses on computer networking and distributed systems. He can be contacted at chankhun@usc.edu.

Peter B. Danzig received his B.S. in Applied Physics from the University of California Davis in 1982 and his Ph.D in Computer Science from the University of California Berkeley in 1989. He is currently an Assistant Professor at the Univerity of Southern California. His research addresses both building scalable Internet information systems and flow, congestion and admission control algorithms for the Internet. He has served on several ACM SIGCOMM and ACM SIGMETRICS program committees and is an associate editor of *Internetworking: Research and Experience*. He can be contacted at danzig@usc.edu.

Charels J. Neerdaels received his BAEM in Aerospace Engineering and Mechanics in 1989, from the University of Minnesota, Minneapolis. After several years work in the defense industry, he continued his education in Computer Science at the University of Southern California. He has recently left the University to become a Member of Technical Staff, Proxy Development at Netscape communications, and can be reached at chuckn@netscape.com. CA 94043

Michael Schwartz received his Ph.D in Computer Science from the University of Washington in 1987, after which time he joined the faculty of the Computer Science Department at the University of Colorado - Boulder. Schwartz' research focuses on international-scale networks and distributed systems. He has built and experimented with a dozen information systems, and chairs the IRTF Research Group on Resource Discovery, which built the the *Harvest* system. Schwartz is on the editorial board for *IEEE/ACM Transactions on Networking*, and was a guest editor of *IEEE Journal of Selected Areas in Communication*, for a 1995 Special Issue on the Global Internet. In 1995 Schwartz joined @Home (a Silicon Valley startup doing Internet over cable), where he is currently leading the directory service effort. Schwartz can be reached at schwartz@home.net.