Squid and ICP: Past, Present, and Future

Duane Wessels
wessels@nlanr.net

August 6, 1997

Abstract

Hierarchical Web caching has become prevalent throughout the Internet. The popular Squid software has played an important role. We present a short history of Squid and the Internet Cache Protocol, and offer our thoughts regarding its future evolution.

1 Introduction

Ever since the World-Wide Web rose to popularity around 1994, much effort has focused on reducing latency experienced by users. Surfing the Web can be slow for many reasons. Server systems become slow when overloaded, especially when hot spots suddenly appear. Congestion can also occur at network exchange points or across links, and is especially prevalent across trans-oceanic links that often cost millions of dollars per month.

Caching has proven a useful technique for improving end user experienced latency on the Web. The fundamental concept is the intermediate storage of copies of popular Web documents close to the end users. Caching is effective because many Web documents are requested (much) more than once[1]. Web browsers have local disk caches because individuals often browse the same pages repeatedly. Additionally, there is likely overlap in the set of documents requested by a large group of users. These users can benefit from a shared network cache.

Taking the proxy caching concept one step further, we might like to connect sets of caches together in a hierarchy. A group of Web caches can benefit from sharing another cache in the same way that a group of Web clients benefit from sharing a cache. In a simple cache hierarchy, a set of *child* caches share a common *parent* cache. Child caches forward to their parents requests for objects they do not have.

The Internet Cache Protocol (ICP) provides support for informed selection of a next-hop cache, including implicit indications of network congestion. Although the diagnostic functionality of ICP may be useful, simple cache topologies do not really require ICP to operate; other approaches can help recognize and deal with network failures. ICP is especially suited to enabling *sibling* relationships (i.e. queries between pairs of child caches). A group of sibling caches effectively acts as a large, distributed cache, a particularly useful configuration when no organization is willing to operate a parent cache for a large community with no explicit revenue stream for it.

The difference between sibling and parent relationships is in their role during cache misses: parents can help resolve misses, but siblings must not. Caches should not forward requests to sibling caches unless they know the sibling has the desired object. Without a mechanism to ascertain whether a cache has a given document, sibling relationships would not even be possible. ICP provides this functionality.

In this article we assume the reader is already familiar with HTTP/1.0[2] and at least aware of HTTP/1.1[3]. For additional information on ICP, please see our Internet Drafts (soon to be RFCs)[4, 5] and other work[6].

2 Squid

2.1 History

Squid was developed as a follow-on to one component of the Harvest project[7], the latter effort originally funded in 1994 by the U.S. Advanced Projects Research Agency (ARPA). Harvest focused on developing tools for effective use of Internet information, with emphasis on indexing and searching the Web. The project team developed a multi-faceted architecture for efficient cataloging and dissemination of popular information. One architectural component aimed at minimizing end user response time during Web navigation was an *object cache*, a platform strategically placed within the network topology that would store copies of popular documents to save users from having to wait for transmission of the document from the original source. The object cache component was named Harvest *cached*. The other Harvest components (the *gatherer* and the *broker*) were quite popular as well, and are still being used in a number of places. In fact, a group of volunteers has emerged to continue development of that code.

At that time, the only other Web caching (sometimes also called *proxy*) software in widespread use was the CERN HTTP Daemon[8]. The Harvest cache had two features that made it an attractive alternative to CERN's proxy. First, while CERN used a separate process for each request, Harvest serviced almost all proxy requests in a single process. The exception is FTP transfers, which are handled in an external process due to the complexity of the FTP protocol. Second, the Harvest cache software supported the notion of cache hierarchies.

The single-process nature of Harvest was attractive because many cache operators saw their machines slow to a crawl during busy periods. The operating system would spend a lot of time and resources forking and scheduling new processes, a price paid for the simplicity of the CERN server design. Harvest took this tradeoff in the opposite direction. As a single process, it has to worry about many operating system level tasks, in particular memory and file descriptor management.

Note, for the remainder of this article, unless stated otherwise, *Harvest* refers to the cache component of the Harvest research project. It does not in any way refer to the commercial software, which is also known as NetCache.

In late 1995 and early 1996, many members of the Harvest project moved on to industry jobs, bringing the project to a premature close. At the same time, Hans-Werner Braun, k claffy, and Michael Schwartz received U.S. National Science Foundation grant funding for their proposal to deploy a prototype Web caching system within the U.S. In March of 1996 I went to work for k and Hans-Werner on this project.

Shortly thereafter, Peter Danzig, primary technical architect for the original Harvest cache software itself, formed a company to sell a commercial version of the Harvest cache. In order to support the recently NSF-funded cache deployment and research project, we needed to have open and publicly available Web cache software. Since I had already worked for a year on the Harvest project, it made the most sense to take the last pre-commercial version of Harvest and enhance it to support research investigations. To avoid confusion between these two derivations of the Harvest software, we named our software Squid. We have spent the last year incorporating code contributions, suggestions, new feature requests, patches, error reports and other feedback into the Squid software.

After a few months of beta testing, the first official version of Squid (1.0.0) was released in July of 1996. Some of the more significant additional features and enhancements are described below.

2.1.1 Support for If-Modified-Since

The Harvest research code does not support If-Modified-Since (IMS) requests. IMS requests are used when clients or caches have a copy of an object and want to make sure it is the most up-to-date version. Harvest simply ignored the IMS header which caused user agents to always receive full replies. The primary reason that Harvest did not support IMS is because the last modification time is not kept as metadata in memory. To implement this, Squid needs to parse the reply headers as the object was read in from disk. Much of the initial code was written by Henrik Nordstrom.

In Squid 1.0, IMS is only supported between Squid and the client. Squid forwards IMS headers for cache misses, and provides the client with the corresponding reply, but does not update its own state with information from any *Not Modified* replies. Squid 1.0 also never initiates an IMS request on its own.

2.1.2 Public and Private objects

Harvest 1.4 treats all objects it knows about as *public*. Multiple clients can simultaneously receive data as an object is being retrieved. Although often a huge savings for large objects, under certain conditions this approach might allow a second client to receive something that should have been given only to the first. For example, consider a request that includes some authentication information. Because this data requires authentication, other clients should not have access to it from the cache. Instead they should separately authenticate themselves with the origin server.

The implementation decisions allowing this vulnerability are:

- A single hash table is used to index all objects in the cache, including pending objects.
- The hash table key is simply the URL for GET requests.

One reason for keying on only the URL is that the cache uses the URL extracted from an ICP reply to look up the pending request and continue the thread of execution. During the time interval between sending the ICP queries and receiving the replies, additional clients could attach themselves to the reply stream. This is a problem because the reply headers may indicate that the object should only be given to the first client.

Harvest could likely have avoided this problem by using a separate hash table for pending requests, or alternatively by using the *Reqnum* field of the ICP message. In fact, Harvest always set the Reqnum field to zero in ICP replies.

Squid implements the Reqnum field properly and uses it to support both *private* and *public* objects. Private objects are accessible only to the client originating the request, and public objects are available to all clients. Squid indexes objects in the storage hash table with a key that includes an integer number prepended to the URL string. Squid places this number into the Reqnum field of outgoing ICP query messages, and a peer must use the same Reqnum value in its reply. This technique allows Squid to use a cache key so that pending requests can be located from the ICP replies, but not by new clients. All requests start out as private, and remain so during the peer selection stage. Upon receipt of the HTTP reply headers, the object will become public, unless the reply indicates otherwise. Only public objects remain in the cache; private ones are removed immediately after transfer. If Squid is configured with an old Harvest peer (which sets the Reqnum field to zero), then the private object feature must be disabled, because it will be unable to locate the pending requests from Harvest's ICP replies.

2.1.3 Metadata reloading in the background

Harvest 1.4 does not serve any requests until it has rebuilt the cache metadata from the swap log file. Shutting the cache down cleanly will allow a much faster reload upon restarting the cache, similar to the benefit of `fsck` on Unix filesystems. Depending on the cache size, a fast reload could take anywhere from one to ten minutes. A slow reload could take hours. In either case, it is unrealistic to deny service to end users while the cache rebuilds.

Henrik Nordstrom implemented a background processing task to reload the swap log while the cache was serving requests. During this phase, Squid gives `select()` a zero timeout value. Upon a timeout, Squid processes a small number of lines from the swap log.

2.1.4 New Access Control scheme

With Harvest 1.4, the only access control (ACL) mechanism is the client's IP address. While working on Squid, it quickly became apparent that administrators wanted a rich, flexible set of access controls. We implemented the following additional access controls:

- Request method
- Access protocol
- Destination domain name
- Destination IP address
- Port number
- Regular expression matching the URL-path.
- Time of day

However, Squid never postpones requests to make access control decisions. Specifically, the destination IP address is only available if the URL hostname is already present in the IP cache. Squid never delays the ACL decision to wait for a DNS lookup to complete. If the URL contains an IP address instead of a hostname, Squid 1.0 does not make a reverse DNS lookup and compare the answer against the destination domain name.

2.2 Present

The first version of Squid 1.1 was released in December 1996. The most recent (1.1.14) was released July 14, 1997. We describe its new features below.

2.2.1 Refresh Rules and If-Modified-Since

Squid 1.0 removes objects when they expire. Squid 1.1 keeps expired objects on disk and issues If-Modified-Since requests for them. This can result in a significant bandwidth savings since the origin server doesn't need to retransmit the entire object. Unfortunately, IMS does less to reduce latency than bandwidth, because the origin server must still be contacted.

2.2.2 URL Redirector

Several Squid users in the community requested the ability to rewrite requested URLs or perform HTTP redirection. Rather than implement this feature in Squid itself, where an elegant solution satisfying everyone would be difficult, we chose to make it an external process. If configured to use redirection, Squid sends every request to one of the *redirector* processes. The redirector program must be supplied (i.e. written) by the administrator. A redirector process reads URLs on stdin, possibly changes the request, then writes the result to stdout. Squid reads the redirector output and modifies the request data structures.

One possible application of the redirector feature is additional access controls. If the redirector program has access to a list of forbidden URLs, it can redirect the request to a page describing exactly why the original request could not be allowed.

2.2.3 Reverse IP Lookups, client hostname ACLs.

Squid 1.1 includes support for access controls based on client domain name, previously impossible because Squid did not do reverse DNS lookups. Much like the IP cache, Squid caches reverse lookup results in the *FQDN cache*. We note that the domain lookups will incur some additional delay and we thus do not necessarily recommend their use.

2.2.4 Cache directory structure changes

The previous version of Squid used 100 subdirectories under each *cache_dir*. For large caches (e.g., more than 10^6 objects) this design would involve thousands of files per subdirectory. Such large directories can potentially cause unnecessary delays during filesystem operations.

A patch from Mark Treacy modified Squid to use a two-level directory structure, defaulting to 16 first-level directories and 256 second-level directories. In this configuration, a million objects would lead to a more sensible 256 objects per subdirectory.

2.2.5 Network Probe Database

Both Harvest and Squid allow requests to be routed based on the hostnames from URLs. However, there are a few problems with this approach. First, it requires a lot of manual configuration. The cache configuration file must list each domain with each peer; it is only practical to list top-level-domains (TLDs). In addition, domain restrictions don't work for URLs with IP addresses instead of fully qualified domain names.

The biggest problem is that the domain names are unrelated to network topology, apart from the rough mapping provided by the two-letter country code TLDs. These national TLDs can frame only a very coarse Web routing system, and international (top-level) domains, e.g., `.com`, `.net`, are even worse because they have no bearing at all to topology. For these reasons, any routing scheme based on domain names is inauspicious; like any other routing scheme, effective cache-level routing needs knowledge of the underlying network topology.

Squid 1.1 has an optional feature to acquire topology data with ICMP. When the *query_icmp* option is enabled, Squid builds up a table of hop counts and round-trip times for the origin servers it encounters. Squid aggregates this data by IP network under the assumption that two hosts on the same local network will have similar values. Via an external process, Squid caches send and receive ICMP echo requests to server hosts at a rate of no more than once per five minutes. Squid then includes the results of these network probe measurements in ICP reply messages. The cache collecting ICP replies uses the network measurements to select the best peer, ideally the one most toward the origin server.

As a cache collects network measurements from its peers, it adds them to its local table, learning over time which peers are good choices for which sources. The cache will be closer to some sources than any of its peers are; for these it can simply fetch directly and avoid the ICP querying. This approach complicates the peer selection algorithm, since instead of remembering a single best parent, Squid must now remember a list of possible parents until all ICP replies arrive.

2.2.6 Round-Robin IP

Squid's IP cache stores all addresses returned by the `gethostbyname()` function call. However, Squid 1.0 only uses the first address. Popular Web sites such as Microsoft require many servers, so an address lookup on `www.microsoft.com` returns 15–20 IP addresses. If the server at the first address fails, Squid 1.0 would never try any of the other addresses. This behavior is fixed in version 1.1. It cycles through all available addresses and deletes addresses for which TCP connections fail.

2.2.7 Neighbor features (neighbor_type_domain, miss_access).

When building a mesh of caches, having the same peering relationship (parent vs. sibling) for all requests might not be desirable. Consider, for example, a pair of upper-level caches in two different countries. Country A might want to forward requests for country B requests to the cache running in country B, in vice-versa. In other words, cache B would be a parent for B domains, and cache A would be a parent for A domains. But what about other requests? The cache administrators might want to leverage the fact that they are already peering with each other.

When some cache operators in Europe were building a high-level cache mesh, they were thwarted by the fact that the parent or sibling relationship appled to all requests. For example, the German cache wanted to forward requests for `.uk` requests to is neighbor cache in the U.K. At the same time, the German cache wanted to use the U.K. cache as a sibling for all other requests. Squid 1.0 did not support treating a peer as a parent for some requests, and as a sibling for others. Squid 1.1 supports this functionality. By using the the *neighbor_type_domain* configuration option, Squid relationships can depends on the domain name of requested the URL.

Note that an ICP query does not include any parent or sibling designation, so the receiver has no indication of how the peer cache is configured to use it. This issue becomes important when a cache is willing to serve cache hits to anyone, but only handle cache misses for its own customers. In other words, whether to allow the request depends on if the result is a hit or a miss. To support this functionality, Squid acquired the *miss_access* feature in October of 1996.

In addition to being awkward to implement, *miss_access* brings its own complication: it requires that the ICP reply be an extremely accurate prediction of the result of a subsequent HTTP request. This prediction is challenging if not impossible since the ICP request cannot convey the full HTTP request. Additionally, there are more types of HTTP request results than there are for ICP. The ICP query reply will either be a hit or miss, but an HTTP request might result in a *Not Modified* reply from the origin server. Such a reply is not strictly a hit since the peer needed to forward a conditional request to the source. At the same time, its not strictly a miss either since the local object data is still valid, and the Not-Modified reply required from the origin server is quite small.

2.2.8 The NOVM version

Perhaps the most common complaint about Squid is its memory consumption. The two primary uses of memory are for object metadata (approximately 120 bytes per object) and in-transit requests. Squid stores objects in-transit fully in memory until the transfer completes, at which point Squid either writes them to disk or purges them from memory. This design flaw requires significant memory for for a cache with many active clients and many large objects.

To alleviate this problem, we developed a branch version of Squid 1.1 that did not retain in-transit object data in memory. Instead, this *NOVM* version writes all objects to disk as it retrieves them, while cache clients simultaneously read from the disk files. This approach essentially doubles the number of file descriptors required because the swap file is open both for reading and writing as Squid downloads the object. In other words, the NOVM version trades memory for file descriptors. For cache machines without enough physical memory, the NOVM version is a viable alternative.

2.2.9 Multicast ICP

Inspired by ideas and code from Martin Hamilton, Squid 1.1 supports sending ICP queries to multicast addresses. The bandwidth savings of multicast can be significant for large cache meshes. While the ICP queries can be multicasted, ICP replies are always unicast for several reasons:

- Multicasting ICP replies would not reduce the number of packets sent.
- It prevents other group members from receiving unexpected replies.
- The reply should follow unicast routing paths to indicate (unicast) connectivity between the receiver and the sender since the subsequent HTTP request will be unicast routed.

Without multicast, the cache administrator must specify every member of a cache mesh in every cache's configuration file. With multicast, new members could simply voluntarily join the multicast group. Although it is tempting to explore the use of multicast as a way to simplify cache configuration, it also introduces security issues. since joining a multicast group requires no special privilege or authentication, and trusting a given multicast group is not typically a safe assumption. For this reason, Squid still requires that all trusted neighbor caches be specified in the configuration file.

2.3 The Future of Squid

We have begun work on Squid version 1.2, in which we plan to support the following new features. Those which we have not yet implemented we identify with an asterisk (*).

2.3.1 Independent swap directories

Squid versions 1.0 and 1.1 assume that all *cache_dir* directories are of the same capacity. Even worse, the file-number-to-pathname algorithm changes when the number of *cache_dir* directories changes. This means that if you lose one disk, you effectively lose your entire cache.

Version 1.2 makes each *cache_dir* independent, i.e., each directory can be a different size, and have a different number of first- and second-level swap directories. In addition, one can configure swap directories to prevent any new objects being written to that disk ('object-read-only') to facilitate planning in advance for directory removal. Each swap directory has a separate swap log (renamed to *swap.state*) for full independence.

Because all swap directories are equal in version 1.1, Squid allocated swap files in a round-robin fashion. For version 1.2, Squid allocates new swap files to the swap directory that has the greatest percentage of free space.

2.3.2 Asynchronous I/O

Whenever possible, squid uses non-blocking I/O, which really only works for sockets and pipes. Most Unix systems execute disk operations immediately. That is, a `read()` or `write()` never returns *EWOULDBLOCK* for disk files. Other system calls such as `open()`, `close()`, and `unlink()` always complete before returning, and although generally very fast, they do block the calling process for short amount of time. When dealing with a high rate of disk operations, such blocking can adversely affect cache performance.

One way to reduce this burden is to execute the disk operations in separate processes. Squid 1.1 has an external process to unlink cache files. Early on, we tried using the *AIO* library functions available on Irix and Solaris. However, these functions did not integrate well with Squid's lower level disk I/O routines. Normally, Squid would write a linked list queue of blocks to a disk file, each block using its own `write()` call. Using an `aio_write()` instead offered little gain, and there were also minor incompatibilities between Irix and Solaris.

Stewart Forster provided us with a significant patch that implements portable and efficient asynchronous I/O. Rather than relying on the semi-portable `aio_write()` and related functions, Stewart's code needs only a POSIX threads library (*pthreads*). We now also combine multiple write buffers into a single contiguous buffer for more efficient output. In addition to reading and writing, Stewart's code also implements open, close, and unlink asynchronously.

2.3.3 Background validation

Stewart Forster also contributed code to implement background validation of a slow cache rebuild. If Squid shuts down without writing a clean swap log file, the subsequent restart must `stat()` every cache file to verify its existence and correct size. Squid version 1.1, made the `stat()` calls as it read the log file, so it could potentially take many hours to fully read the log file.

Stewart's patch allows Squid to read the log file at full-speed while marking objects for delayed validation. After reading the entire log file, Squid begins the validation of all objects via `stat()`. If Squid receives a request during this period for an object it has not yet validated, it validates that object immediately before satisfying the request.

2.3.4 HTTP/1.1 persistent and pipelined connections

The Web is currently in a transition between HTTP/1.0 and HTTP/1.1. The Apache development group has released an HTTP/1.1 compliant version of their HTTP server. In general we find that HTTP/1.1 features are implemented incrementally, and Squid itself is lagging behind on the HTTP/1.1 front. We have implemented persistent and pipelined client connections, but as of August 1997 have not yet implemented persistent connections to servers or neighbor caches. The World-Wide Web Consortium has reported observing both reduced latency and reduced number of packets transmitted with pipelined requests[9].

2.3.5 FTP requests handled internally

We originally believed that the FTP protocol was too complex to implement inside the cache itself and so used an external module, *ftpget*, which was ugly and difficult to understand. Squid 1.2 no longer uses a separate process, the FTP functionality is fully integrated into the mainline code. Besides finding it easier than we had thought, we were also motivated to integrate because of the increasing number of configuration options Squid needed to pass to *ftpget*. Additionally, this change should make it much easier to implement FTP PUT operations, although we have not done so yet.

2.3.6 Replaced *local_domain* and *inside_firewall*.

The *local_domain* option forces Squid 1.1 to forward certain requests directly to the origin server. Similarly, the *inside_firewall* option prevents Squid from ever forwarding certain requests to origin servers. Minor problems with these two options include:

- They use as selection criteria only the URL hostname or IP address; they cannot differentiate based on the access protocol (e.g. FTP vs. HTTP).
- The names can be misleading. Sometimes people want their cache to never make direct connections, but are not behind a firewall.

Squid 1.2 has re-implemented these features as options *always_direct* and *never_direct*, which use Squid's ACL elements for selection criteria.

2.3.7 Customizable error messages

Squid cache administrators often complained that error messages were hard-coded into Squid's executable, particularly problematic for non-English speaking administrators. Squid now supports fully configurable error messages, using printf-style tokens to substitute relevant parts of a request into the message.

2.3.8 More robust request forwarding*

A potentially frustrating aspect of Squid 1.1's request forwarding is its selection of a single location to forward a request. It might be the origin server, or perhaps a neighbor cache. If the request cannot be forwarded to that destination (e.g. due to connection failure) the user receives an error message. Squid 1.1 does not have a concept of *backup parent*. It does not retry failed TCP connection attempts.

Squid 1.2 will use a more robust scheme, where the administrator can specify an ordered list of options for forwarding cache misses. For example, one might want to specify a scheme to:

1. Query siblings using ICP and select first *ICP_HIT*.
2. Select primary parent cache, with a 5 second connection timeout
3. Select backup parent cache, with a 10 second connection timeout
4. Forward to the origin server

2.3.9 More metadata written to swap log

We now write additional fields to the swap log (*swap.state*). In version 1.1 the last-reference time was lost when Squid reloaded the metadata from disk. Actually, Squid does not write the last-reference time to the swap log; rather it sets the last-reference time to the last-valid time upon restart. Version 1.2 logs separately the last-reference time, the reference count, and the *StoreEntry->flags* values. The reference count keeps the replacement algorithm consistent across restart. The flags support the HTTP/1.1 *Must-Revalidate* feature.

2.3.10 MIME changes

In early 1997 the *squid-users* mailing list carried a great debate about content types, inspired by *ftpget*'s need to include a valid *Content-Type* header for the files it retrieved. *Ftpget* determines content type using the URL filename suffix, which if unknown or non-existent (as in the case of *README*), defaulted to either *text/plain* or *application/octet-stream*. The default content type is set at compile-time with a `#define` macro.

If the default were *text/plain*, the user-agent would display the content in the browser window, undesirable when the object was a binary data file or something else ungracefully viewable. A default of *application/octet-stream* would cause Squid to save the file to disk, also not what you want for README files.

This problem stems from the need to gateway data from an FTP server to an HTTP client. The MIME content-type is a fundamental feature of HTTP. On the other hand, FTP has no concept of content types, so Squid has to guess. Version 1.2 determines the content-type from a table of regular expressions loaded at run-time, allowing administrators to customize their cache to meet their needs.

2.3.11 Callback data registry

Both Harvest and Squid have been prone to numerous memory access bugs over the years. In fact, both packages have special signal-handling code to detect Bus Errors and Segmentation Violations, and mail the administrator with a request to generate a stack trace and send it to the developers. In some cases these errors are, in retrospect, quite obvious and easy to fix. There is another class, however, which is a bit more subtle.

When Squid needs to perform an operation that would normally block the process, it registers a callback function to trigger when the operation completes. The callback function always takes some *callback data* as an argument. For example, since the original Harvest implementation, DNS lookups have always been made by external *dnsserver* processes. When the lookup completed, the callback was made so that the request processing would then continue.

This design assumes the *callback data* memory will still be valid when the callback occurs. If the memory has been freed prior to the callback, the callback function will likely write into memory allocated to some other data structure. Then, when the trashed memory is next accessed (perhaps quite some time later), a fatal Segmentation Violation may occur. Because the cause and effect of such bugs may be separated by minutes or hours in time, they are often difficult to detect.

Squid 1.1 has evolved numerous ways to deal with this problem, in some cases by locking data structures with counters, in others by using *unregister* functions to disable callbacks before the operation completes. Each data structure potentially used as *callback data* needs its own implementation to prevent memory access errors.

Squid 1.2 includes a generic callback data registry, to which callback data pointers are *added*, then *locked* while the blocking operation is in progress, *unlocked* when it completes, and *deleted* when memory is freed. Delete operations that occur while the memory is still locked will be delayed until the lock count reaches zero. Before the callback function is called, Squid verifies the callback data memory with the registry; if marked invalid, Squid unlocks and frees it, and never makes the callback.

0		31
OPCODE	VERSION	PACKET LENGTH
REQUEST NUMBER		
AUTHORIZATION		
SENDER HOST ADDRESS		

Figure 1: The original ICP message format.

2.3.12 Limits on resource utilization*

Resource-conscious administrators have also asked for the ability to limit the resources (mostly bandwidth) consumed by some subset of cache clients. For example, an cache-bearing ISP might want to allow unlimited usage to users on network A, but limit the users of network B to 50 KB/sec each. Unfortunately we have not yet come up with a good, general framework for limiting network resources.

3 ICP

3.1 History

The Harvest research code used the the ICP message format shown in Figure 1.

Note that ICP does not have a field for the request method; the payload consists of only the URL. The early Harvest designers assumed that only GET requests would be cacheable, to there would be no need to use ICP for other request methods. The lack of a request method in the ICP message is not a severe deficiency, since in practice non-GET requests are indeed not cached.

It would be nice to have the request method in the ICP message when we try to use ICP to select an appropriate next-hop cache. Consider an Intranet cache with multiple parent caches located on the firewall border, one of which it must select to forward the request since the internal cache cannot directly contact the origin server. For GET requests we can use ICP, but for a POST request we must use other criteria. Normally one of the parent caches will be statically designated as the next-hop cache for all non-GET requests.

The original ICP design had opcodes for supporting multiplexed object transport over TCP: *ICP_SEND*, *ICP_SENDA*, *ICP_DATABEG*, *ICP_DATA*, and *ICP_DATAEND*. The SEND and SENDA opcodes support object requests, similar to HTTP's GET. The three DATA opcodes support transmission of object data back to the requester[10]. In theory these opcodes serve a single, persistent TCP connection between neighbor caches. Objects would be multiplexed by sending them in small chunks with an ICP header to identify them.

There may have been a very early implementation of these opcodes in Harvest. However, around the time of Harvest 1.4, these features were removed in order to make the code simpler.[1] Since then, ICP supports only cache querying, while HTTP handles all object transport.

Harvest never implemented Authorization in ICP, so the the eight-octect *Authorization* field remained unused. In fact, no specific authorization techniques were discussed; the field was only a placeholder for future work. This unused space turned out to be very useful for future versions, but not for authorization.

Harvest always set the *Reqnum* field to zero. In theory, this field would be a unique number for every ICP query, and replies copy the value of *Reqnum* into the reply. Squid utilizes this field to support private and public objects. The fact that Harvest 1.4 always set Reqnum to zero caused some incompatibilities in the two ICP implementations. To accommodate, Squid disables private objects if it finds a Harvest neighbor that fails to set the *Reqnum* field properly.

[1] This explains why there is a gap between Squid's default HTTP port (3128) and the ICP port (3130). Originally Harvest used port 3129 for this non-standard cache-to-cache communication channel.

<table>
<tr><td>0</td><td></td><td>31</td></tr>
<tr><td>OPCODE</td><td>VERSION</td><td>PACKET LENGTH</td></tr>
<tr><td colspan="3">REQUEST NUMBER</td></tr>
<tr><td colspan="3">OPTIONS</td></tr>
<tr><td colspan="3">OPTION DATA</td></tr>
<tr><td colspan="3">SENDER HOST ADDRESS</td></tr>
</table>

Figure 2: The ICP message format used by Squid 1.1

The design of the Harvest research code made ICP very easy to implement. Recall that Harvest always removed expired objects, so if an object was found in the cache, it was known to be valid. Because Harvest did not support IMS requests, the ICP header has no corresponding IMS field.

3.2 Present

In Squid, the ICP message format is essentially the same as it has always been. As shown in Figure 2, the unused authorization field offers support for new and optional features. *Options* is a bitfield of option flags, and *Option Data* is used to send additional information. There are only two options presently defined.

1. The HIT_OBJ option indicates support for the HIT_OBJ opcode (described below).
2. The SRC_RTT option supports the network probe database described earlier. When this option bit is set in an ICP query, the reply should include the peer's estimated network RTT to the origin server, if available. The RTT estimate is placed in the lower half of the *Option Data* field.

3.2.1 ICP_HIT_OBJ

The HIT_OBJ option was inspired by the presence of many small Web objects amenable to piggybacking atop ICP replies, avoiding the three-way handshake and other TCP overhead of the subsequent HTTP request. Using HIT_OBJ offers strong potential for bandwidth savings.

One problem with HIT_OBJ is the larger UDP packet size, increasing the likelihood of fragmentation and reassembly, which also implies an increased chance of being dropped due to congestion. Packet loss is particularly bad for ICP since it incurs the two-second timeout.

Another problem is that HIT_OBJ replies require more time to generate. Whereas caches can return standard hit and miss replies immediately, returning an HIT_OBJ reply requires loading the object data into memory first. HIT_OBJ replies may thus be among the last to arrive, and are more likely to arrive after the two-second timeout. In some cases the tradeoff may not be worth it (avoiding the TCP connection versus losing the hit reply). In a cache hierarchy it may be better to have a normal ICP HIT rather than no hit at all.

We were originally quite enthusiastic about this feature, until we began to hear about associated complications, derived from the fact that the ICP request cannot convey everything from the HTTP request. In particular, the ICP query does not retain information from the HTTP/1.1 'Cache-control: Max-age' header. Some cache operators were noticing that HIT_OBJ was delivering older-than-requested objects. Another example is the 'Cookie' header, sometimes used to deliver customized content. For these reasons we now recommend against using HIT_OBJ.

3.2.2 ICP_MISS_NOFETCH

We added a new opcode named *ICP_MISS_NOFETCH* to assist Squid during the reload phase. Originally called *ICP_RELOADING*, this opcode allows a cache to tell peer caches to "go away, I'm busy" while reloading metadata from disk. The primary motivation was so that the reload process could complete more quickly. When a cache receives the *ICP_MISS_NOFETCH* reply, it knows that its peer is alive, but that it does not want to service the request at that time. Note, Squid still returns *ICP_HIT* messages during the reload phase.

This opcode proved to be very useful in another situation. If a parent cache notices that a significant number of its requests fail, it assumes there is a network problem on its default path to the Internet. The cache then essentially disables itself for a short period of time by returning *ICP_MISS_NOFETCH* instead of *ICP_MISS*. Again, it continues to return *ICP_HIT*'s because those requests can be satisfied locally.

3.2.3 Max-Age problems

One serious problem for cache hierarchies is mismatched freshness parameters. Consider a cache C using strict freshness parameters so its users get maximally current data. C has a sibling S with less strict freshness parameters. When an object is requested at C, C might find that S already has the object via an ICP query and an *ICP_HIT* response. C then retrieves the object from S.

In an HTTP/1.0 world, C (and C's client) will receive an object that was never subject to its local freshness rules. Neither HTTP/1.0 nor ICP provides any way to ask only for objects less than a certain age. If the retrieved object is stale by C's rules, it will be removed from C's cache, but it will subsequently be fetched from S so long as it remains fresh there. This configuration miscoupling problem is a significant deterrent to establishing both parent and sibling relationships.

HTTP/1.1 provides numerous request headers to specify freshness requirements, which actually introduces a different problem for cache hierarchies: ICP still does not include any age information, neither in query nor reply. So S may return an *ICP_HIT* if its copy of the object is fresh by its configuration parameters, but the subsequent HTTP request may result in a cache miss due to any `Cache-control:` headers originated by C or by C's client. Situations now emerge where the ICP reply no longer matches the HTTP request result.

In the end, the fundamental problem is that the ICP query does not provide enough information to accurately predict whether the HTTP request will be a hit or miss. In fact, the ICP Internet Draft[4] is vague on this subject. What does ICP HIT really mean? Does it mean "I know a little about that URL and have some copy of the object?" Or does it mean "I have a valid copy of that object and you are allowed to get it from me?"

3.3 Future

The future of ICP is far from clear. Administrators seem to either like or hate it. One argument against ICP is that request forwarding incurs additional delay, whose magnitude depends on the network path between neighbor caches. It is very undesirable to use ICP over a high-latency, high-loss path.

Unfortunately, Squid greatly relies on ICP as a failure-detection mechanism. Its essentially as if Squid is afraid to contact a peer cache until it receives an ICP reply indicating the peer is alive and well. Squid does have other considerably less satisfactory methods of detecting and dealing with failures. Squid makes only one attempt to forward any request: if it fails, the user receives an error message. Fixing this behavior is described in the previous section and is on our todo list for version 1.2.

Some people believe that ICP is not necessary at all, regardless of network characteristics between neighbor caches. They believe that a wide-area, loose association of Web caches is not a viable long-term model for Web caching on the Internet. Instead, leaf-node caches will establish a single parent relationship with the cache operated by their Internet service provider. Wide-area cache meshes are seen by some as unnecessary because it is likely to be just as fast to fetch directly from the source. It is difficult to say for certain if these beliefs are true. We believe that when Squid acquires an improved and more robust request forwarding scheme, ICP will be less frequently used for parent relationships and perhaps almost exclusively used between sibling caches.

Another important issue is the 'Max-Age problem' described earlier, which requires that ICP messages somehow include the relevant HTTP header fields. Likely candidates would be *If-Modified-Since*, *Max-Age*, and *Last-Modified*. With these headers, sibling Squid caches could have very different refresh parameters and still interoperate well with each other. But would they be sufficient? We believe they would not.

HTTP requests may have additional headers that may affect the decision to return a hit or a miss. One obvious such header field is the Entity Tag (ETag), used to support an alternate method of cache validation, instead of using IMS. An ETag is an opaque, variable-sized string that identifies a resource. Essentially a cache asks a server if the supplied ETag identifies the current version of the named resource. The *Accept-Language* request header may also affect the hit/miss determination, as well as others we have not named or even invented yet.

Because the HTTP protocol is quite rich and complex, it is not realistic to expect that future versions of ICP can still use a fixed, binary message format, unfortunate because processing of the current format is fast and efficient. A future binary format could conceivably include extensible fields for future enhancements, but we risk it being as complicated and resource-intensive as parsing a standard CRLF-delimited HTTP request. Therefore, we plan to develop an implementation of ICP which uses full HTTP requests in query messages. Such a scheme will certainly provide us with more accurate ICP replies. Of course if Squid cannot process the new format quickly enough, administrators could fall back to the current method if desired.

4 Conclusion

While caching has been integrated into the topologies of major U.S. providers such as @Home, MicroSoft, Digex, AOL, and an increasing number of smaller ISPs, it is still beset with research questions desperately needing attention. As the software framework for the NLANR hierarchical caching prototype, Squid and ICP will likely continue to have important roles in the exploration of these questions and advancment of Web caching techniques in general.

We admit openly that thus far it has been difficult to quantify the costs and benefits of ICP usage, or even cache usage at all, particularly in the U.S. where bandwidth is not yet prohibitively expeensive. Factors influencing the economics of deploying caches include: bandwidth costs, administrative costs, user support issues, topology / cache hierarchy considerations, request rates, user base (e.g., more useful if homogeneous), and occurrences of hot sets and thrashing. It seems likely that the deployment of charging schemes based more closely on actual resource consumption costs will emerge, providing greater incentive for use of the cache. We hope Squid and related analysis utilities can continue as a flexible platform for exploring the costs and benefits of caching, as formidable a research challenge as that is in the current climate.

Other research challenges include: delta encoding for transmission of changed material (vs. retransmissions of entire content); building PICs awareness into proxies; developing multicast communications among clustered caches, automating cache discovery/selection; benchmarking proxy servers; establishing effects of ICP and TCP implementations on cache performance and the implications of HTTP 1.1 on cache consistency; push technologies; nearest server location; easing configuration burden while maintaining security, and satisfying economic models.

We also expect Squid and ICP will play a role in more short term operational considerations: growing optimal cache topology configurations, i.e., levels of hierarchy, placement in the infrastructure, load balancing, clustering, and requisite redundancy; preferred cache configurations, i.e., RAM, disk space, refresh rates, and link speeds; and usage/performance statistics.

Finally, Squid and ICP are evolving to support more powerful policy: handling cache busting and striping of cookies; privacy considerations associated with logs (particularly customer usage patterns); copyrights and related issues of 'implied consent' for caching by virtue of posting materials on the Internet; content filtering and PICs labeling.

Squid serves as a reference implementation of hierarchical cache/ICP functionality, and may become a core component of the Internet middleware infrastructure. We hope that by maintaining support for this neutral code platform, the community will continue to be able to more easily and flexibly study this wide range of issues. Thus far we have been inspired by the enthusiastic response.

5 Acknowledgements

Over relatively short time period that Squid has been in use, we have been extremely grateful for the feedback and ideas received from the user community. Many people have made significant contributions to our efforts, many more than we have been able to mention here. You know who you are. We thank you.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing Reference Locality in the WWW," in *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, (Miami Beach, Florida), December 1996.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transport protocol – HTTP/1.0," *Network Working Group RFC 1945*, May 1996. `http://ds.internic.net/rfc/rfc1945.txt`.

[3] R. Fielding *et al.*, "Hypertext transport protocol – HTTP/1.1," *Network Working Group RFC 2068*, January 1997. `http://ds.internic.net/rfc/rfc2068.txt`.

[4] D. Wessels and K. Claffy, "Internet cache protocol (ICP), version 2," *draft-wessels-icp-v2-03.txt*, July 1997. `http://ds.internic.net/internet-drafts/draft-wessels-icp-v2-03.txt`.

[5] D. Wessels and K. Claffy, "Application of internet cache protocol (ICP), version 2," *draft-wessels-icp-v2-appl-02.txt*, July 1997. `http://ds.internic.net/internet-drafts/draft-wessels-icp-v2-appl-03.txt`.

[6] D. Wessels and k claffy, "ICP and the squid web cache." `http://www.nlanr.net/%7ewessels/Papers/icp-squid.ps`, July 1997.

[7] Bowman, Danzig, Hardy, Manber, Schwartz, and Wessels, "Harvest: A scalable, customizable discovery and access system.," Tech. Rep. CU-CS-732-94, Department of Computer Science, University of Colorado, August 1994. `ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Jour.ps.Z`.

[8] A. Lutonen, H. F. Nielsen, and T. Berners-Lee, "Cern httpd," July 1996. `http://www.w3.org/pub/WWW/Daemon/Status.html`.

[9] H. F. Nielsen *et al.*, "Network performance effects of http/1.1, css1, and png," June 1997. `http://www.w3.org/pub/WWW/Protocols/HTTP/Performance/Pipeline.html`.

[10] P. Danzig *et al.*, "Internet cache protocol specification 1.4," September 1994. `http://excalibur.usc.edu/icpdoc/icp.html`.