Configuring Hierarchical Squid Caches

Duane Wessels, wessels@nlanr.net August 19, 1997

Configuring Web caches in a hierarchy or mesh can be a daunting process. Caching software such as Squid offers many configuration options. We explain the implications for joining a cache mesh and discuss when certain features will or will not provide benefits. We present common-case scenarios for Internet service providers, corporations, and educational institutions. We also explain how to understand and process cache log files to determine how your cache is interacting with the mesh.

Contents

1 Background **3**

1.1 HTTP . . . 3

1.2 Web Proxies and Caching . . . 4

1.3 Hierarchical Caching . . . 4

2 Terminology and Concepts for Hierarchical Caching **4**

2.1 Cache . . . 4

2.2 Objects . . . 4

2.3 Hits and Misses . . . 5

2.4 Origin Server . . . 5

2.5 Hierarchy vs. Mesh . . . 5

2.6 Neighbours, Peers, Parents, Siblings . . . 5

2.7 Fresh, Stale, Refresh . . . 5

3 Internet Cache Protocol **5**

4 The Basic Neighbour Selection Process **6**

5 Squid's Features for Hierarchical Caching **7**

5.1 *local_domain* (v1.0.0) . . . 7

5.2 *local_ip* (v1.0.0) . . . 8

5.3 *cache_host_domain* - Routing by Domains (v1.0.0) . . . 9

5.4 *cache_host_acl* - More Domain Routing (v1.0.0) . . . 9

5.5 *query_icmp* - Probing the Network with ICMP (v1.1.9) . . . 9

5.6 *proxy-only* (v1.0.0) . . . 10

5.7 *icp_hit_stale* (v1.1.11) . . . 11

5.8 *no-query* (v1.0.0) . . . 11

5.9 *default* Parent (v1.1.0) . . . 11

5.10 *round-robin* Parents (v1.1.6) . . . 12
5.11 Parent weights (v1.0.0) . . . 12
5.12 *neighbour_timeout* (v1.0.0) . . . 13
5.13 *inside_firewall* (v1.0.0) . . . 13
5.14 *firewall_ip* (v1.0.0) . . . 13
5.15 *hierarchy_stoplist* (v1.0.0) . . . 14
5.16 *neighbour_type_domain* (v1.1.0) . . . 14
5.17 *miss_access* (v1.1.0) . . . 15
5.18 *source_ping* (v1.0.0) . . . 16
5.19 *single_parent_bypass* (v1.0.0) . . . 16
5.20 Non-ICP Neighbours (v1.0.0) . . . 16
5.21 Multicast ICP (v1.1.6) . . . 16
5.21.1 Sending multicast queries . . . 17
5.21.2 Receiving multicast queries . . . 17
5.21.3 Receiving replies from multicast group members . . . 17
5.21.4 Counting multicast group members . . . 18
5.22 ICP_HIT_OBJ (v1.0.0) . . . 18
5.23 Failure Ratio (v1.0.16) . . . 19

6 Neighbour Selection Details 19
6.1 Step One - Do we need to look up the IP address? . . . 20
6.2 Step Two - Selecting the Next-Hop . . . 22
6.3 Up/Down Status . . . 25

7 Common Configurations for Leaf Caches 25
7.1 General . . . 25
7.2 One parent, no Siblings . . . 26
7.3 Multiple Parents - Load Balancing . . . 26
7.4 Multiple Parents - Request Routing . . . 27
7.5 Siblings, no Parents . . . 27
7.6 Both Siblings and Parents . . . 28

8 Common Configurations for Parents and Siblings 28
8.1 Being a Parent . . . 28
8.2 Being a Sibling . . . 29
8.3 Running a Cluster of Caches . . . 29
8.4 Mutual Parents . . . 29

9 Analyzing Squid's *access.log* **30**
9.1 Cache Result . . . 30
9.2 HTTP Status . . . 31
9.3 Hierarchy Code . . . 31
9.4 Hierarchy Categorization . . . 33
9.5 The NLANR Caches . . . 34
9.5.1 Hierarchy Analysis By Category . . . 34
9.5.2 Hierarchy Analysis By Cache . . . 34
9.6 Checking on your Siblings . . . 38

10 The Cache Manager **38**
10.1 Server List . . . 38
10.2 Client List . . . 39
10.3 Network Probe Database . . . 41

11 About This Document **42**
11.1 Revision . . . 42

1 Background

1.1 HTTP

The Hypertext Transfer Protocol (HTTP) is used to transfer data between Web servers (e.g. Apache) and Web clients (e.g. Netscape Navigator). Currently, the majority of clients and servers use *HTTP/1.0*. Although early and partial, *HTTP/1.1* deployment has already begun. HTTP typically comprises a large fraction of Internet traffic; at times up to 80% of the traffic on some wide-area backbone links in the U.S.

An HTTP request is comprised of three major parts: a request method, a Uniform Resource Locator (URL), and a set of request headers. An HTTP reply consists of a numeric result code, a set of reply headers, and an optional reply body. The most common request method is GET, which is a request to retrieve the resource named by the URL. A GET request is essentially a *download* operation. Another commonly used request method is POST, which is essentially an *upload* operation.

A special kind of GET request is a *conditional GET*, distinguished by its use of an *If-Modified-Since* header, and whose reply depends upon the modification date of the resource compared to the date provided in this *If-Modified-Since header*. For brevity, we'll refer to this as an IMS request. IMS is an important feature for caching, as it can save a lot of bandwidth and latency by having the server send only a small *Not Modified* response if the client already holds a current copy of the resource. If the resource has changed since the IMS timestamp, the cache receives a new version from the server.

The HTTP/1.1 *Cache-control* header allows both clients and servers to "override the default caching algorithms." We are primarily interested in the *Max-age* directive, which lets the client place an upper limit on how old an object can be and still satisfy the request without refreshing the document from the source. Here *age* refers to the elapsed time since the origin server provided the data. If the cached object is older than the client requires, then the cache must forward the request to the origin server.

1.2 Web Proxies and Caching

World-Wide Web caches are implemented as *proxies*. Normally Web clients (browsers) make direct connections to Web servers. However, client browsers may instead be configured to connect to a proxy server, which then forwards the request on behalf of the client. For this reason, Web caching is also occasionally referred to as *proxy caching*, although not all proxies are caches. Sometimes proxies function only as a gateway between two sides of an Internet firewall, and do not retain copies of any objects in the process. Caching implies proxying, but proxying does not imply caching.

Because the proxy is passing Web requests from servers to clients, there is strong benefit for the proxy to *cache* popular requests locally. The proxy users benefit because they receive cached documents more quickly. Additionally, the organization itself benefits by reducing the amount of traffic imposed on its wide-area Internet connection. By deploying Web caches, an organization will typically reduce its HTTP traffic by 30–50%.

1.3 Hierarchical Caching

Cache Hierarchies are a logical extension of the caching concept. 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. However, there are some disadvantages as well. Whether the advantages outweigh the disadvantages probably depends on the specific situation. The main advantages are:

- Additional cache hits. In general, you can expect that another 10% requests received by your cache would be cache hits in your neighbour caches.
- Request routing. By routing requests to specific caches you may be able to direct HTTP traffic along a certain path. For example, if you have two paths to the Internet, one of which is cheap and the other being expensive, you can send HTTP traffic over the cheap link.

Among the disadvantages:

- Configuration hassles. Configuring neighbour caches requires coordination of both parties, a burden that exacerbates as membership becomes large.
- Additional delay for cache misses. Whether or not utilizing neighbour caches improves end-user performance depends on many factors, e.g., delays between peers, link congestion, and whether or not ICP is used.

2 Terminology and Concepts for Hierarchical Caching

2.1 Cache

In this document, the term *cache* refers to an HTTP proxy that caches some requests.

2.2 Objects

An *object* is a generic term for any document, image, or other type of data available on the Web. Uniform Resource Locators (URLs) identify objects unambiguously, and can refer to data available from HTTP, FTP, Gopher, and other types of servers. Because a much of today's Web content is comprised of images, audio, video and binary files, we prefer to say that the cache stores objects rather than *documents* or *pages*.

2.3 Hits and Misses

A *cache hit* means a valid copy of the requested object exists in a cache. A *cache miss* means that the object does not exist, or is no longer valid. A cache must *forward* cache misses toward the origin server.

2.4 Origin Server

The *origin server* is the authoritative source for an object. For URLs, the origin server is simply the hostname part of the URL.

2.5 Hierarchy vs. Mesh

Web caches can be arranged hierarchically, or in a mesh. When the cache topology has a tree-like structure, we usually use the term *hierarchy*. If the structure is rather flat, we call it a *mesh*. In either case these terms simply refer to the fact that caches can be "connected" to each other.

2.6 Neighbours, Peers, Parents, Siblings

The terms *neighbour* and *peer* are synonymous, and generically refer to other caches in a hierarchy or mesh. The terms *parent* and *sibling* refer to the relationship between a pair of caches.

In a parent relationship, the *child* cache will forward requests to its parent cache. If the parent does not hold a requested object, it will forward the request on behalf of the child. A cache hierarchy should closely follow the underlying network topology. Parent caches should be located along the network paths towards the greater Internet. For example, if your Internet Service Provider (ISP) operates a cache, it should probably be a parent to yours, since your Web traffic will have to travel along your ISPs infrastructure anyway.

With a *sibling* relationship, a peer may only request objects already held in the cache; a sibling can not forward cache misses on behalf of the peer. The sibling relationship should be used for caches "nearby" but not in the direction of your route to the Internet. For example, it may make sense for a number of department-specific caches within an organization to have sibling relationships among them. This approach is even more compelling when there is no parent cache available for the organization as a whole.

Note that a single peering relationship is one-way. A child can forward requests to its parent, but not vice-versa, unless another peering relationship exists for the other direction.

2.7 Fresh, Stale, Refresh

These terms refer to the status of cached objects. If an object is *fresh* then it will be returned as a cache hit. If the object is *stale*, then Squid will *refresh* it by including an IMS request header and forwarding the request on toward the origin server.

3 Internet Cache Protocol

The Internet Cache Protocol was developed as a fundamental part of the Harvest project. ICP's role is to provide a quick and efficient method of intercache communication, offering a mechanism for establishing complex cache hierarchies. ICP allows one cache to ask another if it has a valid copy of a named object, thereby improving its chances of selecting a neighbor cache that will return a cache hit. In addition, ICP allows requests to flow among child caches.

Harvest and Squid also utilize ICP to provide an indication of network conditions. Failure to receive an ICP reply likely means the path is severely congested, severed, or the peer host is down. In any case, we would not want to forward a request to that peer, if possible. Similarly, the order in which ICP replies are received can indicate which ones are "closer" and/or less loaded, and therefore likely to give better performance.

ICP *messages* are transmitted as UDP packets. One advantage to UDP is that the code becomes easier to implement because each cache needs to maintain only a single UDP socket. Also, as alluded to above, we normally want ICP messages to be dropped in the face of congestion. One disadvantage is that ICP becomes useless on highly congested links, perhaps exactly where caching is needed most.

Another disadvantage is that ICP adds some delay to the processing of a request. The extra delay is almost exclusively due to the transmission time of the UDP packet. In other words, a request will be delayed by the maximum round-trip time to your neighbour caches. Because of this delay, ICP is not appropriate for all situations.

ICP can also function as a conveyor of cache access policies. Rather than forward a HTTP request, only to have the peer deny it, ICP will give us advance warning that the HTTP request would be denied. In that case, we certainly would not forward the request to that peer. However, this does not mean that we should query all peers for all requests and let the access controls decide which will be allowed. The cache administrators should communicate their terms-of-use to each other. Squid will complain both when it sends too many ICP_DENIED replies to a neighbour, and when it receives too many ICP_DENIED replies from a neighbour.

The ICP messages are generally very small. They consist of a 20-octet binary header followed by a URL string. The median ICP message size seen by the NLANR caches is 66 bytes. In a cache hierarchy, there will usually be many more ICP messages than HTTP requests. Depending on the type of relationship (parent or sibling) the ratio of ICP to HTTP may be anywhere from 2 to 10.

As of this writing two Internet Draft documents on ICP are going through the final stages of becoming Informational RFCs. The draft documents are named `draft-wessels-icp-v2-03.txt` and `draft-wessels-icp-v2-appl-03.txt`.

4 The Basic Neighbour Selection Process

Before we describe Squid features for hierarchical caching, we first briefly explain the neighbour selection process, aspects of which have motivated features described in the following section.

First, what does "neighbour selection process" refer to? When Squid is unable to satisfy a request from the cache (i.e. its a cache miss), Squid must decide where to forward the request next. There are basically three choices: a parent cache, a sibling cache, or the origin server. Squid uses ICP to make this decision.

1. Squid sends an ICP query message to its neighbours (both parents and siblings). The ICP query includes the requested URL and is sent in a UDP packet. Squid remembers how many queries it sends for a given request.

2. Each neighbour receives its ICP query and looks up the URL in its own cache. If a valid copy of the URL exists, the cache sends ICP_HIT, otherwise an ICP_MISS message.

3. The querying cache now collects the ICP replies from its peers. If the cache receives an ICP_HIT reply from a peer, it immediately forwards the HTTP request to that peer. If the cache does not receive an ICP_HIT reply, then all replies will be ICP_MISS. Squid waits until it receives all replies, up to two seconds. If one of the ICP_MISS replies comes from a parent, Squid forwards the request to the parent whose reply was the first to arrive. We call this reply the FIRST_PARENT_MISS. If there is no ICP_MISS from a parent cache, Squid forwards the request to the origin server.

We have described the basic algorithm, to which Squid offers numerous possible modifications, including mechanisms to:

1. Send ICP queries to some neighbours and not to others.
2. Include the origin server in the ICP "pinging" so that if the origin server reply arrives before any ICP_HITs, the request is forwarded there directly.
3. Disallow or require the use of some peers for certain requests.

We further describe these options below.

5 Squid's Features for Hierarchical Caching

Squid has evolved in many ways with additional features to enhance the neighbour selection algorithm described above. Most of these features have been suggested by the user community. In this section we describe the configuration file directives that are relevant to hierarchical caching and the neighbour selection algorithm.

After each option we indicate in parentheses the Squid version where it first appeared. As of late August 1997 the most recent Squid release is version 1.1.15.

5.1 *local_domain* (v1.0.0)

In most situations, there will be some set of servers which are "local" to the Web cache. Because these servers are nearby, we should contact them directly instead of forwarding the request to another cache. Squid allows you to configure local servers by domain name to avoid sending their requests up the cache hierarchy. For example:

```
local_domain sample.com
```

Squid checks the hostname part of requested URLs. If the hostname ends with *sample.com*, Squid forwards the request directly to the origin server. All of these URLs would match the above specification:

```
http://www.sample.com/
http://ftp.uk.sample.com/
http://sample.com/foo
```

However, these URLs would not match:

```
http://www.freesample.com/
http://www.nlanr.net/stats/sample.com/
```

Also note that if the URL had the IP address for *sample.com* instead of the domain name, it would NOT match. The *local_ip* option, described below, can be used for this latter case.

You may specify multiple domains on one configuration line, or multiple configuration lines:

```
local_domain    foo1.com foo2.com foo3.com
local_domain    bar1.com
local_domain    bar2.com
local_domain    bar3.com
```

Cache administrators often do not want to cache the objects from local servers. The *local_domain* directive does not prevent this caching; you need to use the *cache_stoplist* directive instead.

5.2 *local_ip* (v1.0.0)

In some situations it might be easier to specify local servers by IP address instead of domain name, which Squid supports with the *local_ip* directive. *local_ip* works for both IP addresses and domain names in requested URLs.

One drawback to *local_ip* is that Squid must look up the IP address before deciding if a given request is local or not, which adds the DNS lookup delay to the request processing. Thus we must recommend the use of the *local_domain* directive whenever possible. Of course, if you really need need to catch the case when the URL in question uses an IP address, then you should use *local_ip* as well. These two directives may be used together; not just one or the other.

There are three basic ways to specify IP address values. First, zeros can be used to indicate the wildcard octets of a network. For example:

```
local_ip 10.0.0.0
local_ip 172.16.0.0
local_ip 192.1.2.0
local_ip 10.7.7.0
local_ip 172.16.1.1
```

Note that we do not assume classful IP addresses. If all octets are nonzero, then it specifies a single IP address. The second way is with CIDR prefix notation. For example, the above lines may be equivalently written as:

```
local_ip 10.0.0.0/8
local_ip 172.16.0.0/16
local_ip 192.1.2.0/24
local_ip 10.7.7.0/24
local_ip 172.16.1.1/32
```

Of course the CIDR notation allows more flexibility, because you can specify any prefix length, not just 8, 16, 24 or 32. Finally, the more traditional netmask notation is accepted as well:

```
local_ip 10.0.0.0/255.0.0.0
local_ip 172.16.0.0/255.255.0.0
local_ip 192.1.2.0/255.255.255.0
local_ip 10.7.7.0/255.255.255.0
local_ip 172.16.1.1/255.255.255.255
```

The *local_ip* addresses can be preceded with a '!' to negate the meaning. For example,

```
local_ip 10.0.0.0/8
```

means that any address in the 10.0.0.0 network is local, and

```
local_ip !10.0.0.0/8
```

means that every address is local, EXCEPT those in the 10.0.0.0 network. This is most often useful when a more-specific subset of a network is not local. Such as:

```
local_ip !172.16.4.4 172.16.0.0
```

5.3 *cache_host_domain* - Routing by Domains (v1.0.0)

Often, it will be necessary to restrict the requests sent to neighbour caches. In other words, we want to force some requests to certain neighbours, and/or prevent requests to certain other neighbours. One motivation might be to distribute load among a set of neighbours. We can specify which domains a given cache can handle with the *cache_host_domain* directive. This directive is not required, so by default Squid assumes that a neighbour cache can handle all requests.

As an example, consider four cooperating caches, which we will abbreviate as *cache1* through *cache4* (Normally we would use fully qualified domain names). These caches are dedicated to caching URLs in the `.au` top-level domain, but rather than all caches holding all objects, they have been partitioned. *cache1* holds `net.au` URLs, *cache2* holds `com.au` URLs, *cache3* holds `edu.au` URLs, and *cache4* holds all the other URLs in the `.au` domain.

Now, if we need to configure a cache that will act as a child of this cluster of four, we would specify the partitioning as follows:

```
cache_host_domain       cache1  net.au
cache_host_domain       cache2  com.au
cache_host_domain       cache3  edu.au
cache_host_domain       cache4  !net.au !com.au !edu.au au
```

Note that we can either list the domains in a positive or negative manner. Preceding a domain with '!' means that the querying cache can NOT use the specified cache for the corresponding domain.

Squid searches the domain list in the order give, and stops searching as soon as it finds a match, or reaches the end of the list. For this reason, you must list more-specific domains before less-specific ones.

5.4 *cache_host_acl* - More Domain Routing (v1.0.0)

In some cases, simple domain-based restrictions may not be sufficient For example, what if you wanted to send HTTP requests to one neighbour and FTP requests to another?

You can also restrict neighbour caches based on Squid's general access control features. To accomplish the example above, we could specify it thusly in the config file:

```
acl FTP proto FTP
acl HTTP proto HTTP
cache_host_acl cache1 FTP
cache_host_acl cache2 HTTP
```

This configuration would cause Squid to send only FTP requests to *cache1* and only HTTP requests to *cache2*. As with *cache_host_domain*, you can negate ACL elements:

```
cache_host_acl cache2.foo.net !FTP
```

which would send everything except FTP requests to *cache2*.

Note that use of the ACLs might cause additional delays. Some of the ACL types require forward or reverse DNS lookups.

5.5 *query_icmp* - Probing the Network with ICMP (v1.1.9)

Routing by domain name suffers from a couple of problems:

1. Domain names don't match network topology.
2. Poor scaling; we can't really list more than top-level-domains (TLDs).
3. Some TLDs can't even get you in the right country (com, net, org).

These reasons motivated the *query_icmp* feature in Squid and ICP. Each cache keeps a table of ICMP round trip time (RTT) measurements to the origin servers. The measured values can be returned in the ICP reply messages. Squid uses the RTT values to select the neighbour which is "closest" to the origin server. If the local cache itself is closest, then the request is forwarded directly to the origin server.

This feature is not enabled by default and requires the cooperation of both parties in a peering relationship. First of all, your neighbour caches need to be configured to make ICMP measurements. They will return these measurements in their ICP replies to your cache. Your cache will use the measurements to select the best neighbour.

To get Squid to make the ICMP measurements, Squid must be compiled with a special option. The **USE_ICMP** option must be enabled in *src/Makefile*, and then Squid recompiled. After Squid has been built, the *pinger* program must be installed with root privileges. *pinger* is responsible for sending and receiving the ICMP messages and relaying them to Squid. Unix systems require root privileges to send and receive ICMP, so we use an external process rather than make all of Squid run with superuser privileges. The commands below indicate the sequence of events required to enable this feature and install the *pinger* program. The Squid FAQ has more details on this procedure.

```
% vi src/Makefile
% make clean all
% make install
% su
# make install-pinger
```

Now that we have your neighbours collecting ICMP measurements, we must tell them to send them to you. To do this, enable the *query_icmp* option in your configuration file.

```
query_icmp on
```

There are two ways to see how this is working. You should see some requests logged as CLOSEST_PARENT_MISS in the *access.log* file. Also, if your Squid program is compiled with the **USE_ICMP** option, then you can view the Network Probe Database from the cache manager interface. This database will include both the measured ICMP RTTs from your cache and your neighbour caches.

When your Squid cache is making ICMP measurements, you can configure it to forward directly to origin servers which are estimated to be some small number of router-hops away. The *minimum_direct_hops* directive sets this value. Note that the hop-count estimation is not always perfect, but it should be safe to set this parameter to 4 or 5.

5.6 *proxy-only* (v1.0.0)

The *proxy-only* option appears on a *cache_host* line, and specifies that Squid should not save a local copy of any object retrieved from that cache. This feature is often useful in a cluster of sibling caches to prevent each cache from holding every object. When the caches are close to each other (e.g. on the same ethernet segment), then it costs relatively little to transfer an object from one to the other. Specify this option with:

```
cache_host cache1 sibling 3128 3130 proxy-only
cache_host cache2 sibling 3128 3130 proxy-only
```

There is no requirement that one use *proxy-only* exclusively with sibling caches; it also works fine for a parent cache.

Note that the *proxy-only* feature can generate some confusing error messages for some versions of Squid. When Squid initiates a transfer from a neighbour with the *proxy-only* option set, Squid puts the object into "delete behind" mode. That is, Squid deletes the data as it is delivered to the client. If the user aborts the transfer before it completes, Squid will log the error message:

```
ERR_NO_CLIENT_BIG_OBJ: http://www.nlanr.net/
```

As of Squid v1.1.11, this error message will no longer appear for *proxy-only* requests.

5.7 *icp_hit_stale* (v1.1.11)

The description for *proxy-only* above alludes to creating a larger, distributed cache from a cluster of caches with sibling relationships. The *proxy-only* option makes sure that an object is stored in only one of the caches. However, its not quite that simple.

Recall that an ICP_HIT reply indicates that a subsequent HTTP request from the querying cache would result in a cache hit. This feature is critical for siblings because if the HTTP request turns out to be a cache miss, we would be violating the sibling relationship. For this reason, we only return ICP_HIT if the object is cached and *fresh*. In fact, to be safe, we require that the object be fresh for the next 30 seconds. If the object will be stale soon, we return an ICP_MISS instead.

This procedure implies that the distributed cache idea will not work as we would like. Once an object becomes stale in one of the caches, a sibling cache will never be able to refresh the object because an ICP_MISS reply must be returned. This situation is unfortunate, since the distributed cache model was a primary early motivation for the sibling relationship. To have the distributed cache concept work, siblings must return ICP_HIT replies even for stale objects. Squid supports the capability with the *icp_hit_stale* configuration option:

```
icp_hit_stale on
```

Enabling this option allows your your sibling caches to refresh stale objects through your cache. In fact, it changes the peering relationship to be something between the strict sibling and parent definitions. If *icp_hit_stale* is enabled, you must NOT also use the *miss_access* feature.

5.8 *no-query* (v1.0.0)

The *no-query* option appears on a *cache_host* line and prevents Squid from sending ICP queries to a specified neighbour cache. It is most useful with the *default* or *round-robin* options, described below. An example configuration is:

```
cache_host cache1 parent 3128 3130 no-query default
```

5.9 *default* Parent (v1.1.0)

Recall that in the basic neighbour selection algorithm, Squid uses a parent only if the parent sends it an ICP reply. This limits Squid in at least two situations:

1. If no ICP replies arrive, we might still want (or need) to use a parent cache. This is particularly important if the cache is behind a firewall.

2. We might not want to use ICP at all, but still forward requests to a parent cache. If there is only one parent cache, the choice is obvious. But what about when there are multiple parents?

The *default* option designates a parent cache as a default choice in the absence of other indications (such as ICP replies). For example:

```
cache_host cache1 parent 3128 3130 default
```

And if we did NOT want to use ICP at all, we would use:

```
cache_host cache1 parent 3128 3130 default no-query
```

Note that when searching for a default parent, the *cache_host_domain* restrictions still apply. Thus, it is not unreasonable to have multiple default parents, but it makes little sense to have a default sibling. Squid searches the neighbour caches in the order given in the configuration file, and uses the first parent with the *default* designation (if it passes the *cache_host_domain* restrictions).

5.10 *round-robin* Parents (v1.1.6)

The *round-robin* option is similar to *default*, except that Squid forwards the request to the parent with the lowest use count. The *cache_host_domain* restrictions still apply, of course. A typical configuration might look like:

```
cache_host cache1 parent 3128 3130 round-robin no-query
cache_host cache2 parent 3128 3130 round-robin no-query
cache_host cache3 parent 3128 3130 round-robin no-query
```

Squid treats all *round-robin* parents equally. It is not currently possible to, e.g., forward 25% of the requests to one parent and 75% to another.

5.11 Parent weights (v1.0.0)

Recall that the basic neighbour selection algorithm selects the FIRST_PARENT_MISS reply in the absence of any ICP_HIT replies. So with two parents, one close (low RTT), and the other far away (high RTT), Squid will typically use the close parent, although in certain situations this choice may not actually be optimal.

Squid allows you to balance this out a little bit by specifying weighting values for each parent. It does not make any sense to assign weights to a sibling (Do you know why?). Squid calculates the round-trip time between sending an ICP query and receiving the corresponding reply, and divides this RTT by the weighting factor. A higher weight will artificially lower the calculated RTT between peers, thereby favoring it in the selection algorithm. You specify weights as follows:

```
cache_host cache1 parent 3128 3130 weight=10
cache_host cache2 parent 3128 3130 weight=4
cache_host cache3 parent 3128 3130 weight=1
```

When all your parent caches have similar (unweighted) round-trip times, this feature gives you the ability to increase the chance of selecting one peer over the others, simply by giving it a higher weight. Some people use this option to configure a primary parent and a backup parent.

5.12 *neighbour_timeout* (v1.0.0)

In describing the basic neighbour selection algorithm we referred to a two second timeout. The actual value is configurable with the *neighbour_timeout* directive. If you have a high latency link and want to use ICP across it, you may want to increase this value. However, in that case we would probably recommend NOT using ICP at all. To increase the timeout to four seconds, you would write:

```
neighbour_timeout 4
```

5.13 *inside_firewall* (v1.0.0)

Firewalls present some interesting problems for Web caches. The cache must be able to figure out if a given server is inside or outside of the firewall. Squid has two options to configure this. *inside_firewall* is a list of DNS domains which are inside of the firewall (and can be contacted directly if needed). All other domains are assumed to be outside of the firewall and therefore only reachable by forwarding the request to a parent cache.

As with the *cache_host_domain* directive, the '!' negation operator may precede a domain name, and more-specific domains should be specified first.

Please keep in mind that this directive is used for caches *behind* a firewall. If the cache is *on* the firewall then it should be able to directly connect to external systems, and firewall configuration lines should be unnecessary.

A sample configuration is:

```
inside firewall !extern.sample.com sample.com
```

Why do we need to specify `extern.sample.com`? Because otherwise Squid will think it is *inside* the firewall and able to connect to it directly. Squid assumes that any domain not matching the *inside_firewall* list is *outside* the firewall.

The special keyword *none* can be used in place of a domain name:

```
inside_firewall none
```

means that all hosts are outside of the firewall and must be requested via a neighbour cache.

People will often make use of the *inside_firewall* directive when there is really not a firewall in place. They want to force all requests to go through a parent cache, and never to make direct connections to origin servers. To make matters worse, Squid has not been designed to be very persistent in forwarding requests. Squid likes to choose a single, best server to send the request to, and it will try that server only once. If the connection or request fails for some reason, Squid returns an error message.

This behavior can be particularly frustrating with firewall configurations. If Squid thinks all of its neighbours are down, or fails to receive any ICP replies, it will return a message saying that it is unable to satisfy the request. When the administrator goes to check things out, it usually works just fine, thereby increasing the frustration. We are addressing this significant deficiency for the upcoming version of Squid.

5.14 *firewall_ip* (v1.0.0)

It is also possible to specify a set of IP addresses inside the firewall with the *firewall_ip* option. As with the *local_ip* option, this normally requires an IP address lookup before forwarding the request, so *inside_firewall* is preferred. The IP address specifications here are exactly the same as for *local_ip*, including the ability to precede an address with '!' to indicate it is *outside* the firewall.

Note that the *firewall_ip* list must be exhaustive. That is, if a given address matches the *firewall_ip* list, then it is *inside* the firewall, otherwise it is *outside*.

5.15 *hierarchy_stoplist* (v1.0.0)

The *hierarchy_stoplist* allows you to specify URLs that Squid should never forward to neighbour caches, if possible (the firewall directives take precedence). It is somewhat similar in function to *local_domain*, however, it applies to the URL as a whole. The *hierarchy_stoplist* is simply a list of strings which we check for in URLs.

It makes sense that we should only use the cache hierarchy for requests likely to result in cache hits. To reduce the load placed on parent caches, if we can identify requests that we know will not result in a cache hit, we should prevent forwarding them to parents.

The default Squid configuration file sets the *hierarchy_stoplist* to

```
hierarchy_stoplist cgi-bin ?
```

That is, cgi-bin and query requests (indicated by the question mark) are unlikely to ever result in a cache hit. Some may want to change *cgi-bin* to simply *cgi* to catch even more of those nasty CGI programs.

5.16 *neighbour_type_domain* (v1.1.0)

Normally the parent or sibling relationship applies to the neighbour cache as a whole. However, some hierarchical configurations need to define the relationship at a finer granularity. Consider, for example, a top-level cache for the country of Germany, intended as a service to two groups:

1. Users (or other caches) within Germany may use this cache to improve their access to the Internet as a whole.
2. Other caches outside of the country may use this cache to improve access to servers within Germany.

Should a cache in another country specify the German cache as a parent or as a sibling? Normally we would want it to be a *parent* for the `.de` domain, since any `.de` request can be forwarded to it.

In addition to accepting all `.de` requests from external caches, the German cache is also willing to serve any cache hits (but not misses) to anyone. This latter description defines the sibling relationship. What we need, therefore, is the ability to have a parent relationship for some requests, and a sibling relationship for others.

The *neighbour_type_domain* directive fills this need:

```
cache_host sibling cache.foonet.de 3128 3130
neighbour_type_domain cache.foonet.de parent de
```

Now we will treat cache.foonet.de as a sibling for most requests, and as a parent for `.de` requests. You probably only need to specify the domains when the relationship is the opposite of the default. Of course it is also possible to have a parent relationship by default, and a sibling relationship for specific domains:

```
cache_host parent cache.sample.com 3128 3130
neighbour_type_domain cache.sample.com sibling .com .net
neighbour_type_domain cache.sample.com sibling .au .de
```

The effect here is that when we receive an ICP_MISS from *cache.sample.com*, we would never forward the request there if the origin server is in the `com`, `net`, `au`, or `de` domains. However, we can forward requests for URLs in any other domains to *cache.sample.com* because it would be considered a parent cache.

5.17 *miss_access* (v1.1.0)

Note that an ICP query does not include any parent or sibling designation, so the receiver really 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 or not depends on if the result is a hit or a miss. By default Squid does not deny the request based on the hit/miss status. We added the *miss_access* directive to support this type of access control.

In addition to being a somewhat 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 because the ICP query cannot convey the full HTTP request. Additionally, there are more types of HTTP request results than there are for ICP. The ICP 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 from the origin server is quite small.

To use the *miss_access* feature, you first must define a set of ACL elements, and then combine them on the *miss_access* lines with *allow* or *deny* designations:

```
acl Browsers src 10.0.0.0/8
acl Kids 172.16.0.0/16
miss_access allow Browsers
miss_access allow Kids
miss_access deny all
```

In this simple example, all of our end users (Browsers) are on one network (10.0.0.0), and we also have some child caches on another network (172.16.0.0). Of course we allow end users to make any request, since the cache is really for them. Similarly, child caches are all allowed to make any request, including cache misses. However, no one else may request cache misses.

Note that you only need to use *miss_access* if some other cache is using yours as a sibling. Any request to your cache must first pass the normal *http_access* controls, and unless you are someone's sibling, there is absolutely no reason to allow cache hits and deny cache misses. If Squid denies a request for failure to pass the *miss_access* controls, the end user is shown an error message.

A rather serious problem arises when sibling caches use *miss_access* and have differing *refresh_pattern* rules. Again, the fundamental problem is that an ICP query cannot fully express the HTTP request, specifically the Cache-Control parameters. Without those parameters, we cannot accurately predict the hit/miss status in the ICP reply. Occasionally a sibling cache will return ICP_HIT, but the following HTTP request will be a cache miss due to the Cache-Control headers. If *miss_access* is enabled, Squid generates the *miss_access* error message, which will make no sense to either the cache administrator or the end user. If *miss_access* is NOT used, then the sibling cache will forward the request on, which strictly speaking, is a violation of the sibling relationship. How frequently this violation occurs depends on the extent to which the refresh rules differ.

For reasons of backward compatibility, we have not changed the ICP message format to fix this problem. We feel that the best solution to this problem is to configure both siblings with identical refresh parameters. Otherwise, we would recommend against the use of *miss_access*.

5.18 *source_ping* (v1.0.0)

Squid also allows the origin server to be included in the selection algorithm. This is accomplished by sending an ICP_SECHO message to the echo service of the origin server. The SECHO reply is treated like an ICP_HIT in the selection algorithm. Thus, when the origin server is closer than any of the neighbour caches, the SECHO reply will be received first, and the request will be forwarded there directly.

Unfortunately, this feature is often unusable because system administrators have disabled their UDP echo service, or even worse, they log the echo packets, track down the source, and interrogate the responsible party. This became a real problem after CERT issued some warnings regarding denial-of-service attacks to the UDP echo port. Nonetheless, this feature remains available, and to enable it, the configuration file should include:

```
source_ping on
```

An alternative to UDP echo is to use ICMP instead. When Squid is compiled with **USE_ICMP** (described above for *query_icmp*), then ICMP will be used for source ping messages. The only change is that we sneak these messages in as ICMP where they are less likely to be noticed.

5.19 *single_parent_bypass* (v1.0.0)

This option skips the ICP query when the only neighbour to which a request could be forwarded is a parent cache. It is very likely that the ICP reply will be either ICP_HIT or ICP_MISS, both of which would result in selecting that parent. If there are sibling caches involved, then the *single_parent_bypass* setting does not come into effect. The functionality of this option has been superseded by the *no-query* and *default* options, so it is rarely used. To enable this option:

```
single_parent_bypass on
```

5.20 Non-ICP Neighbours (v1.0.0)

Squid allows non-ICP caches (e.g. CERN) to be included in the basic neighbour selection algorithm. Rather than sending ICP query messages, we instead send the special ICP_DECHO message to the UDP echo port of the neighbour cache. The DECHO reply is treated as an ICP_MISS in the selection algorithm.

There are three problems with this feature. First, we must treat the reply as a miss because the neighbour cache does not support ICP. The reply is primarily used to indicate the state of the path between the pair of caches. Second, we are sending to the host's echo service. This really only tells us that the host is up, and not that the cache application is necessarily running. Finally, as described above, paranoid system administrators often disable UDP echo on their hosts, making this technique unusable.

To use this feature, simply specify the ICP port as 7 in a *cache_host* line:

```
cache_host cache.sample.com parent 8080 7
```

Note that it does not make any sense to have a non-ICP sibling cache. (Do you know why?)

5.21 Multicast ICP (v1.1.6)

Recent versions of Squid support sending ICP queries via multicast. This feature also requires the cooperation of both you and your neighbour cache administrators. You must configure your cache to send multicast queries and your neighbour must configure theirs to receive them.

Multicast, although it sounds promising, may not be a good idea for everyone. First of all, you need multicast connectivity to your peers, typically meaning an Mbone tunnel to your site. The best way to test your mbone connectivity is with the *mtrace* program, which is available from *the Xerox PARC FTP site* `<ftp://ftp.parc.xerox.com/pub/net-research/ipmulti/>`.

The primary advantage to multicast is that it can reduce the number of times an ICP packet traverses a single path. The primary disadvantage is that the Mbone infrastructure is sometimes unstable. If a tunnel goes down, you may find that your neighbour caches are not being utilized.

You should be a little bit careful when selecting a multicast group address and TTL parameters. Note that there are no special privileges required to join a multicast group, so anyone can join and "snoop" on your ICP messages. Also, multicast packets are routed based on address, not port number. This means that if you happen to choose an address already in use by a different application, your ICP packets will be sent to the hosts running the other application, and their packets will be sent to your host. There are two good ways to limit the scope of multicast traffic. The simplest is with TTLs, and the other is known as administrative scoping. The multicast TTL scheme is too complex to go into here, but *mtrace* can tell you the minimum required TTL needed between a source and destination. We also recommend you use administrative scoping if possible. For details see the *mrouted* manual page, or your router (probably Cisco) documentation. Also, NLANR has been delegated a block of multicast address for use exclusively by Web caching applications. If you would like to use one of these addresses, just *let us know* `<mailto:nlanr-cache@nlanr.net>`.

5.21.1 Sending multicast queries

To configure Squid to send ICP queries to a multicast address, you need to create another neighbour cache entry specified as *multicast*. For example:

```
cache_host 224.9.9.9 multicast 3128 3130 ttl=64
```

224.9.9.9 is a sample multicast group address; DNS names may be used here as well if available. *multicast* indicates that this is a special type of neighbour. The HTTP-port argument (3128) is ignored for multicast peers, but the ICP-port (3130) is very important. The final argument, *ttl=64* specifies the multicast TTL value for queries sent to this address. It is probably a good idea to increment the minimum TTL by a few to provide a margin for error and changing conditions.

5.21.2 Receiving multicast queries

Your neighbours will need to be configured to receive your multicast queries. To accomplish this, they must listen for packets on the multicast group address by using the *mcast_groups* configuration directive.

```
mcast_groups  224.9.9.9
```

5.21.3 Receiving replies from multicast group members

You must also specify which of your neighbours will respond to your multicast queries, since it would be a bad idea to implicitly trust any ICP reply from an unknown address. Note that ICP replies are sent back to *unicast* addresses; they are NOT multicast, so Squid has no indication whether a reply is from a regular query or a multicast query. To configure your multicast group neighbours, use the *cache_host* directive and the *multicast-responder* option:

```
cache_host cache1 sibling 3128 3130 multicast-responder
cache_host cache2 sibling 3128 3130 multicast-responder
```

Here all fields are relevant. The ICP port number (3130) must be the same as in the *cache_host* line defining the multicast peer above. The third field must either be *parent* or *sibling* to indicate how Squid should treat replies. With the *multicast-responder* flag set for a peer, Squid will NOT send ICP queries to it directly (i.e. unicast).

5.21.4 Counting multicast group members

Recall that in the basic neighbour selection algorithm, Squid waits for all ICP replies to arrive before forwarding the request. This characteristic in problematic for multicast since we send only one query and receive numerous replies. If Squid overestimates the number of replies it will receive, then it will often suffer the two second timeout. If Squid underestimates the number of replies, then it will underutilize its neighbours.

Squid periodically (every 15 minutes) sends out a bogus ICP query to its multicast peers, and tallies the number of replies received before the two second timeout. Squid averages the number of replies with a decay factor so that it adjusts fairly quickly to changing conditions. When waiting for replies to arrive after a normal query, Squid always rounds down the average to the nearest integer value. This counting process is logged in *cache.log* with lines like:

```
97/08/07 13:59:36| Group nlanr.mcast.ircache.net: 6 replies,  6.0 average
97/08/07 14:14:36| Group nlanr.mcast.ircache.net: 5 replies,  5.5 average
97/08/07 14:29:36| Group nlanr.mcast.ircache.net: 4 replies,  5.0 average
```

5.22 ICP_HIT_OBJ (v1.0.0)

The ICP_HIT_OBJ feature was added as another way to reduce latency in cache transfers. Many Web objects are relatively small. For the NLANR caches, the median object size is approximately 3750 bytes. It's an appealing thought to just piggyback such small documents on ICP_HIT replies, thereby avoiding the need for a subsequent TCP connection and HTTP request. We were initially very excited about this feature, but then some problems began to appear.

The fundamental problem is that the ICP query does not convey as much information as an HTTP request. Certain important headers, such as *If-Modified-Since* and *Max-Age*, are not present in the ICP query, which may lead to an ICP_HIT_OBJ reply returning stale data. The ICP query also will not include authentication or identification (i.e. "cookie") headers, which may affect the reply. For these reasons we generally do not recommend use of ICP_HIT_OBJ, and Squid disables it by default.

Prior to Squid v1.0.13, ICP_HIT_OBJ was always enabled. To enable receipt of ICP_HIT_OBJ replies in later versions, use the *udp_hit_obj* configuration option:

```
udp_hit_obj on
```

By default, Squid will send an ICP_HIT_OBJ reply, including the requested object content, if the object will fit into the single UDP reply packet. For most systems the maximum UDP packet size is 12 Kbytes or larger, so an ICP_HIT_OBJ reply can occasionally result in some fairly large UDP packets, which may be undesirable. If the cache administrator prefers the limit the size of the of the ICP_HIT_OBJ message (i.e. the UDP packet), he can specify the maximum allowed ICP packet size with the *udp_hit_obj_size* directive (added in v1.1.4). Remember that the ICP message also includes the 20-octet ICP header and a NULL-terminated URL string. To limit ICP_HIT_OBJ messages to 4 Kbytes, for example:

```
udp_hit_obj_size 4096
```

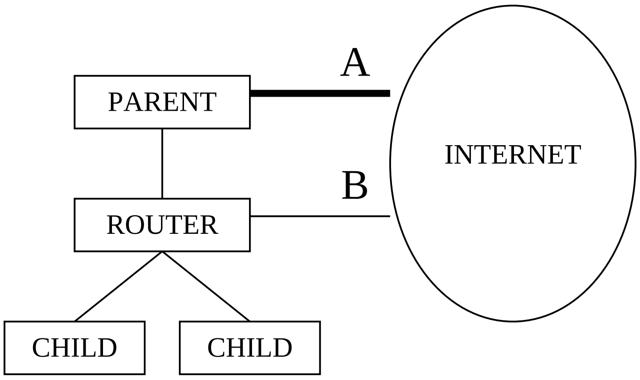

Figure 1: When good parents go bad. Normally, the child cache requests are satisfied via the parent and link A. But when link A fails, the child caches have an alternate path via link B. If the parent cache begins to encounter a large amount of failed requests, it should eliminate itself from the selection process for cache misses.

If you wanted to never send any ICP_HIT_OBJ messages, you could specify a very low, but non-zero value here, for example:

```
udp_hit_obj_size 1
```

5.23 Failure Ratio (v1.0.16)

UDP transport allows ICP to gracefully accommodate network failures, albeit only for failures between a pair of peers. Network failures have also been known to occur between the parent and the rest of the Internet. Consider figure 1 which shows a pair of child caches which have two ways to reach the global Internet, via either link A, or link B. Assume that link A is faster and therefore preferred over B. To make use of link A, the child caches are configured to use a parent cache which routes its traffic over that link.

What happens when link A goes down? The child caches still have good connectivity to the parent, and will therefore receive ICP_HIT or ICP_MISS replies as usual. However, the parent cache will be unable to satisfy any miss requests because its path to the Internet is down. The users of the child caches will get many "connection failed" error messages, even though they have an alternate way of reaching the Internet.

Squid keeps track of its failed requests to cope with this problem. When the ratio of failed to successful requests exceeds a threshold (i.e. 1) then Squid returns ICP_MISS_NOFETCH instead of ICP_MISS replies. This feature allows a parent cache to continue serving hits, but take itself out of the peer selection process for misses.

6 Neighbour Selection Details

For this section, we'll examine the neighbour selection algorithm in detail by running through two functions from the Squid source code. Knowledge of C is not necessarily required.

There are two primary functions to examine. First, Squid calls `protoDispatch()` for all requests to decide if the URL's IP address must be learned. If not, Squid calls the second function, `protoDispatchDNSHandle()` directly. If it does need to look up the address, then `protoDispatchDNSHandle()` is called when the lookup completes.

Note that these functions have evolved considerably since the first Squid versions. As bugs have been pointed out, the code has been rearranged and rewritten in an attempt to accommodate everyone's needs.

6.1 Step One - Do we need to look up the IP address?

First we examine `protoDispatch()`:

```
    protoData->inside_firewall = matchInsideFirewall(request->host);
    protoData->hierarchical = BIT_TEST(entry->flag, HIERARCHICAL) ? 1 : 0;
    protoData->single_parent = getSingleParent(request);
    protoData->default_parent = getDefaultParent(request);
    protoData->n_peers = neighboursCount(request);
```

Here we are assigning some variables for future reference. We check if the origin server is inside our firewall (based only on the domain names listed with *inside_firewall*). The request is 'hierarchical' if it does not match the *hierarchy_stoplist*, and if the request method is GET. The 'single_parent' will be set if the only peer to which this request could be forwarded is a parent cache. Similarly, the 'default_parent' will be set to the first parent cache found with the *default* flag set. The 'n_peers' variable indicates how many neighbour caches would be queried if we decide to use ICP later on. Note that `getSingleParent()`, `getDefaultParent()`, and `neighboursCount()` all use the domain restrictions (i.e. *cache_host_domain*).

```
    if (Config.firewall_ip_list) {
        /* Have to look up the url address so we can compare it */
        protoData->source_ping = Config.sourcePing;
        protoData->direct_fetch = DIRECT_MAYBE;
        protoData->ip_lookup_pending = 1;
        ipcache_nbgethostbyname(request->host,
            fd,
            protoDispatchDNSHandle,
            (void *) protoData);
```

The firewall configuration takes precedence over everything else. First we see if the *firewall_ip* directive is configured. If so, then we need to translate the URL hostname into an address before going any further. At this point we're not sure if we can directly connect to the origin server, so we set 'direct_fetch' to *maybe*. The neighbour selection process will continue in the `protoDispatchDNSHandle()` function when the DNS lookup completes.

```
    } else if (!protoData->inside_firewall) {
        /* There are firewall restrictions, and this host is outside. */
        /* No DNS lookups, call protoDispatchDNSHandle() directly */
        protoData->source_ping = 0;
        protoData->direct_fetch = DIRECT_NO;
        protoDispatchDNSHandle(fd,
            NULL,
            (void *) protoData);
```

If we have reached this code segment then there is no *firewall_ip* list, but *inside_firewall* may still be configured. If we know the request is *not inside* the firewall, then we cannot contact the origin server directly. We must forward the request to a neighbour cache. We do not need to look up the IP address from the domain name. We call `protoDispatchDNSHandle()` directly to continue the process.

```
    } else if (matchLocalDomain(request->host) || !protoData->hierarchical) {
        /* will fetch from source */
```

```
        protoData->direct_fetch = DIRECT_YES;
        protoData->ip_lookup_pending = 1;
        ipcache_nbgethostbyname(request->host,
            fd,
            protoDispatchDNSHandle,
            (void *) protoData);
```

If the request matches the *local_domain* list, or if this is a non-hierarchical request, we will connect to the origin server directly. First we look up the IP address.

```
    } else if (protoData->n_peers == 0) {
        /* will fetch from source */
        protoData->direct_fetch = DIRECT_YES;
        protoData->ip_lookup_pending = 1;
        ipcache_nbgethostbyname(request->host,
            fd,
            protoDispatchDNSHandle,
            (void *) protoData);
```

Recall that above we counted the number of neighbours that we would query. If the result is zero, then we will directly connect to the origin server. The 'n_peers' value will be zero if the domain restrictions do not yield any appropriate neighbours for this request, or if Squid thinks the neighbour caches are down. Below we describe how Squid decides if a neighbour is down.

```
    } else if (Config.local_ip_list) {
        /* Have to look up the url address so we can compare it */
        protoData->source_ping = Config.sourcePing;
        protoData->direct_fetch = DIRECT_MAYBE;
        protoData->ip_lookup_pending = 1;
        ipcache_nbgethostbyname(request->host,
            fd,
            protoDispatchDNSHandle,
            (void *) protoData);
```

Here we check the *local_ip_list*, if defined. Before calling `protoDispatchDNSHandle()` we need to look up the IP address.

```
    } else if (protoData->single_parent && Config.singleParentBypass &&
        !(protoData->source_ping = Config.sourcePing)) {
        /* will fetch from single parent */
        protoData->direct_fetch = DIRECT_MAYBE;
        protoDispatchDNSHandle(fd,
            NULL,
            (void *) protoData);
```

Here we look for the opportunity to use the 'single parent' optimization. In order for this to be used, the only peer to which this request can be forwarded must be a parent, and the *single_parent_bypass* configuration option must be enabled. Additionally, we will not use this optimization if the *source_ping* feature is enabled. If these three conditions are true, we do not need to look up the IP address, so Squid calls `protoDispatchDNSHandle()` directly.

```
    } else {
        /* will use ping resolution */
        protoData->source_ping = Config.sourcePing;
        protoData->direct_fetch = DIRECT_MAYBE;
        protoData->ip_lookup_pending = 1;
        ipcache_nbgethostbyname(request->host,
            fd,
            protoDispatchDNSHandle,
            (void *) protoData);
    }
```

Finally, the last option is to query our neighbour caches with ICP. This decision actually occurs in `protoDispatchDNSHandle()`, but we need to look up the IP address here just in the event that we'll contact the origin server later on.

6.2 Step Two - Selecting the Next-Hop

Now we examine the `protoDispatchDNSHandle()` function, called after completion (successful or failed) of the IP address lookup, or if the lookup is unnecessary.

```
    if (protoData->direct_fetch == DIRECT_YES) {
        if (ia == NULL) {
            protoDNSError(protoData->fd, entry);
            return;
        }
        hierarchyNote(req, HIER_DIRECT, 0, req->host);
        protoStart(protoData->fd, entry, NULL, req);
        return;
    }
```

The variable 'ia' holds the result of the DNS lookup. If we are supposed to directly connect to the origin server, and if the DNS lookup failed (indicated by 'ia' being equal to NULL), then we cannot forward this request. If the DNS lookup was successful, then we call `protoStart()` to start the process of connecting to the server.

```
    if (protoData->direct_fetch == DIRECT_MAYBE
        && (Config.local_ip_list || Config.firewall_ip_list)) {
        if (ia == NULL) {
            debug(17, 3, "Unknown host: %s\n", req->host);
            protoData->direct_fetch = DIRECT_NO;
        } else if (Config.firewall_ip_list) {
            srv_addr = ia->in_addrs[ia->cur];
            if (ip_access_check(srv_addr, Config.firewall_ip_list) == IP_DENY) {
                hierarchyNote(req, HIER_FIREWALL_IP_DIRECT, 0, req->host);
                protoStart(protoData->fd, entry, NULL, req);
                return;
            } else {
                protoData->direct_fetch = DIRECT_NO;
            }
        } else if (Config.local_ip_list) {
```

```
            srv_addr = ia->in_addrs[ia->cur];
            if (ip_access_check(srv_addr, Config.local_ip_list) == IP_DENY) {
                hierarchyNote(req, HIER_LOCAL_IP_DIRECT, 0, req->host);
                protoStart(protoData->fd, entry, NULL, req);
                return;
            }
        }
    }
```

This is where we check the origin server's IP address against the *firewall_ip* and *local_ip* lists. If the DNS lookup failed, then we set 'direct_fetch' to *no* which requires that this request be forwarded to a neighbour cache if at all possible.

Otherwise, we first check the *firewall_ip* list. If the address matches the list then the origin server is *inside* the firewall, in which case we will make a direct connection. The *firewall_ip* list must be exhaustive. That is, if the IP address does not match the list, Squid assumes that the origin server is on the *other side* of the firewall. Again we set 'direct_fetch' to *no* to require use of a neighbour cache.

If the address matches the *local_ip* list then we'll contact the origin server directly. If not, we continue:

```
    if ((e = protoData->single_parent) && Config.singleParentBypass) {
        /* Don't execute this block simply because direct == NO, we
         * might have some DOWN peers and still need to ping them */
        /* Only one parent for this host, and okay to skip pinging stuff */
        hierarchyNote(req, HIER_SINGLE_PARENT, 0, e->host);
        protoStart(protoData->fd, entry, e, req);
        return;
    }
```

Above we're using the single parent optimization if possible.

```
    if (protoData->n_peers == 0 && protoData->direct_fetch == DIRECT_NO) {
        if ((e = protoData->default_parent)) {
            hierarchyNote(req, HIER_DEFAULT_PARENT, 0, e->host);
            protoStart(protoData->fd, entry, e, req);
        } else if ((e = getFirstUpParent(protoData->request))) {
            hierarchyNote(req, HIER_FIRSTUP_PARENT, 0, e->host);
            protoStart(protoData->fd, entry, e, req);
        } else {
            hierarchyNote(req, HIER_NO_DIRECT_FAIL, 0, req->host);
            protoCantFetchObject(protoData->fd,
                entry,
                "No peers to query"
                " and the host is beyond your firewall.");
        }
        return;
    }
```

Above, we have reached a code point where we cannot connect directly to the origin server, and there are no neighbours to query ('n_peers' == 0), so we must select a suitable parent cache. First we try a *default* parent. If one is not available, we'll take the first parent cache which Squid believes to be up. Otherwise, we generate an error message.

```
    if (!neighbours_do_private_keys && !protoData->hierarchical &&
        (e = getFirstUpParent(req))) {
        /* for private objects we should just fetch directly (because
         * icpHandleUdp() won't properly deal with the ICP replies). */
        hierarchyNote(req, HIER_FIRSTUP_PARENT, 0, e->host);
        protoStart(protoData->fd, entry, e, req);
        return;
```

This section of code Squid hopefully never uses anymore; it is called only when one of the neighbour caches does not support 'private keys'. Squid can not use private keys if one of its neighbours is a Harvest-1.4 (or earlier) cache. Harvest-1.4 does not properly support the ICP Reqnum field which enables Squid to implement private cache keys.

```
    } else if (protoData->direct_fetch == DIRECT_MAYBE && ia
        && netdbHops(ia->in_addrs[ia->cur]) <= Config.minDirectHops) {
        hierarchyNote(req, HIER_DIRECT, 0, req->host);
        protoStart(protoData->fd, entry, NULL, req);
        return;
```

This is where we check the *minimum_direct_hops* parameter. If our estimated hops is less than or equal to this value, then we forward the request directly and skip any ICP queries.

```
    } else if (neighboursUdpPing(protoData)) {
        /* call neighbourUdpPing and start timeout routine */
        if (entry->ping_status != PING_NONE)
            fatal_dump("protoDispatchDNSHandle: bad ping_status");
        entry->ping_status = PING_WAITING;
        commSetSelect(protoData->fd,
            COMM_SELECT_TIMEOUT,
            (PF) getFromDefaultSource,
            (void *) entry,
            Config.neighbourTimeout);
        return;
    }
```

Here we call the routine which sends ICP queries. If `neighboursUdpPing()` returns zero, then it didn't really send any queries and we should continue execution of this function. Also note that here we start the two-second *neighbour_timeout* alarm.

```
    if ((e = protoData->default_parent)) {
        hierarchyNote(req, HIER_DEFAULT_PARENT, 0, e->host);
        protoStart(protoData->fd, entry, e, req);
    } else if ((e = getRoundRobinParent(req))) {
        hierarchyNote(req, HIER_ROUNDROBIN_PARENT, 0, e->host);
        protoStart(protoData->fd, entry, e, req);
```

If we reach this far, then we have not sent any ICP queries and it is time for desperate selection of an appropriate next-hop cache. If a *default* parent is defined, we'll take it. Otherwise, if a *round-robin* parent is defined we'll take that instead.

```
    } else if (protoData->direct_fetch == DIRECT_NO) {
        hierarchyNote(req, HIER_NO_DIRECT_FAIL, 0, req->host);
        protoCantFetchObject(protoData->fd, entry,
            "No neighbours or parents were queried "
            "and the host is beyond your firewall.");
```

At this point we have decided that none of the peer caches should receive this request. If we also have decided that this request cannot be forwarded to the origin server directly, then we generate an error message.

```
    } else if (ia == NULL) {
        protoDNSError(protoData->fd, entry);
```

Here we know that we are allowed to connect to the origin server, but if the IP address lookup has failed, then we must generate an error message.

```
    } else {
        hierarchyNote(req, HIER_DIRECT, 0, req->host);
        protoStart(protoData->fd, entry, NULL, req);
    }
}
```

Finally, as a last resort, we have chosen to forward the request directly to the origin server and we have an IP address for it, so off we go with `protoStart()`.

6.3 Up/Down Status

Squid tracks the reachability status of its neighbour caches. When using ICP, Squid marks a neighbour as *down* when the neighbour fails to reply to 20 consecutive ICP queries. As soon as a reply is received again, the neighbour is marked *up*.

In addition to ICP, Squid also monitors TCP connections. When a TCP connection to a neighbour fails, Squid marks the neighbour *down* and begins a process to periodically (every 80 seconds) retry a diagnostic TCP connection. When the diagnostic connection succeeds, the neighbour is marked *up*.

So, in order for Squid to consider a neighbour cache as *up*, these two conditions must be true:

1. The last TCP connection must have been successful.
2. There must be fewer than 20 consecutive unacknowledged ICP queries.

7 Common Configurations for Leaf Caches

7.1 General

The following are general guidelines for configuring leaf caches (those with only browsers as clients). We realize that these may not be appropriate for everyone, however.

- *inside_firewall* and *firewall_ip* should only be used for real firewalls.
- *source_ping* should not be used because server administrators find it very annoying.
- *udp_hit_obj* should not be enabled. It will cause your users to receive stale data.

- *hierarchy_stoplist* should have `cgi-bin` and `?` so that your parent cache is not bothered with those requests.
- Avoid using *miss_access* if at all possible.

7.2 One parent, no Siblings

Configuring a single parent cache is relatively straightforward. Most likely, as with any cache configuration, you would want to use the *local_domain* directive to indicate domains which you do not want to forward to the parent cache. We assume that the parent cache lies somewhere along your default path to the Internet (i.e. at your ISP). If the cache is not along your default path, you may want to consider it as a sibling instead of a parent.

For a single parent configuration, we normally do NOT recommend using ICP. In this case, the primary benefit of ICP would be that it can detect congestion and failure of the parent cache. Squid will monitor the TCP connections to the parent cache anyway. When a TCP connection fails, and Squid marks the parent *down*, your cache will forward requests directly to origin servers until the parent cache comes back up. The disadvantage to ICP is the added latency, which may or may not be significant, depending on the RTT to your parent cache.

Sample configuration:

```
local_domain mydomain.net
cache_host cache.isp.net parent 3128 3130 no-query default
```

7.3 Multiple Parents - Load Balancing

Here we consider using multiple parent caches for load balancing. Instead of a single cache, your ISP (or whoever) may choose to have multiple caches for purposes of load balancing or redundancy. From your point of view, these parent caches are all equivalent and it does not particularly matter which one you use for any given request.

There are really two ways to support load balancing: either with ICP or with the *round-robin* flag. The advantage to using ICP is that the least-busy parent cache will likely be the first to send an ICP reply (all other things being equal). The disadvantage, again, is the additional latency. Using the *round-robin* flag avoids ICP delays. Without, ICP Squid will use the TCP connection status to mark parent caches as up or down.

Another disadvantage to not using *round-robin* and not ICP is that the parent cache selected for a given URL is essentially random. For example, `http://www.nlanr.net/` may be forwarded to *cache1* the first time and then to *cache3* the second time. With ICP, your cache would know which of the caches already hold the URL. If the parent caches themselves have sibling relationships between themselves, then this becomes less of an issue. They will check each other's caches before forwarding the request on to the origin server.

Sample configuration with *round-robin*:

```
local_domain mydomain.net
cache_host cache1.isp.net parent 3128 3130 no-query round-robin
cache_host cache2.isp.net parent 3128 3130 no-query round-robin
cache_host cache3.isp.net parent 3128 3130 no-query round-robin
```

Sample configuration with ICP:

```
local_domain mydomain.net
cache_host cache1.isp.net parent 3128 3130
cache_host cache2.isp.net parent 3128 3130
cache_host cache3.isp.net parent 3128 3130
```

7.4 Multiple Parents - Request Routing

Here we consider multiple parent caches, but now for the purpose of request routing. Instead of all parents being equivalent, we now want to restrict which requests are forwarded to which parents. The *cache_host_domain* or *cache_host_acl* directives must be used to define which requests each parent can handle.

For example, lets assume that your ISP has a pair of caches. One will be dedicated to caching URLs from the `.com` domain. The other will hold all other domains. This is easily configured as follows:

```
local_domain mydomain.net
cache_host cache1.isp.net parent 3128 3130 no-query default
cache_host cache2.isp.net parent 3128 3130 no-query default
cache_host_domain cache1.isp.net com
cache_host_domain cache2.isp.net !com
```

7.5 Siblings, no Parents

If you want to have sibling relationships with other caches, then you must utilize ICP. There are two important things to consider in this situation. First, is the "access denied" problem discussed above (5.17). Either the group of siblings need to have the same *refresh_pattern* rules, or they should not use *miss_access*. Not using *miss_access* requires a fair amount of trust between all of the members.

Second, you should avoid sending large numbers of ICP queries for every request. When your cache must wait for a lot of replies, then the probability that at least one of the replies will not arrive becomes significant. If you plan to query every sibling for every request (i.e. no *cache_host_domain* restrictions), then we recommend keeping the group size small, say five to six members at the most.

Sample configuration:

```
local_domain mydomain.net
cache_host cache.isp1.net sibling 3128 3130
cache_host cache.isp2.net sibling 3128 3130
cache_host cache.isp3.net sibling 3128 3130
cache_host cache.isp4.net sibling 3128 3130
```

This might be a good situation to use multicast ICP if possible. Although multicast is not any less complex to configure, it would be less sensitive to the group size.

```
local_domain mydomain.net
cache_host 224.9.9.9 multicast 3128 3130 ttl=32
cache_host cache.isp1.net sibling 3128 3130 multicast-responder
cache_host cache.isp2.net sibling 3128 3130 multicast-responder
cache_host cache.isp3.net sibling 3128 3130 multicast-responder
cache_host cache.isp4.net sibling 3128 3130 multicast-responder
mcast_groups 224.9.9.9
```

The multicast *cache_host* allows your cache to send queries to other group members, and the *mcast_groups* directive allows your cache to receive queries from the others.

7.6 Both Siblings and Parents

If you have both sibling and parent caches, then you will need to use ICP for the siblings at least. Depending upon the RTT to your parent(s), you may want to use ICP for them as well. For a single parent, we would probably recommend this configuration:

```
local_domain mydomain.net
cache_host cache.isp1.net parent 3128 3130 no-query default
cache_host cache.isp2.net sibling 3128 3130
cache_host cache.isp3.net sibling 3128 3130
cache_host cache.isp4.net sibling 3128 3130
```

For any request, this configuration would cause your siblings to be queried first with ICP. If none of the siblings has the URL, the request would be forwarded to your parent cache (if it was believed to be up).

8 Common Configurations for Parents and Siblings

8.1 Being a Parent

When acting as a parent cache for someone else, the first thing to take care of is the access controls. Your child caches need to provide you with their cache hostnames or addresses. You need to communicate to your child cache(s) exactly which requests you will allow. We generally recommend that parent caches do NOT handle CGI and other non-cachable requests. If the child caches use ICP, then it is important to have your ICP and HTTP access controls match. It would be frustrating at best to allow all ICP queries and then deny specific HTTP requests. Finally, do not use *miss_access* for your child caches.

Sample configuration:

```
acl KIDS src cache.isp1.net cache.isp2.net cache.isp3.net
acl CGI urlpath_regex cgi \?
icp_access allow KIDS !CGI
http_access allow KIDS !CGI
...
...
icp_access deny all
http_access deny all
```

We would also recommend that parent caches use the ICMP features of Squid. This means compiling Squid with the **USE_ICMP** option defined in *src/Makefile*. The ICMP measurements can assist your child caches in selecting the optimal parent cache for a given origin server.

There is really no need for parent and child caches to have the same *refresh_pattern* configurations. We usually recommend that parent caches use conservative refresh rules. However, this is not really important since Squid always includes the first cache's freshness requirements in the *Cache-Control: Max-Age* header.

Note that parent caches tend to have lower hit rates than their child caches. This will be especially true if the parent cache has only a few child caches as clients. The hit rate increases with the number of clients, so quite many child caches may be required to achieve reasonable hit rates in the parent.

8.2 Being a Sibling

Access controls are just as important for sibling caches as for parents. In addition to the *http_access* and *icp_access*, you will also need to carefully consider *miss_access*. If you do not trust your sibling peers enough to not use *miss_access*, then every cache should use the same *refresh_pattern* rules. Failure to do so will result in unexpected "access denied" error messages.

You will receive many more ICP queries than HTTP requests from your sibling peers. For every 100 ICP queries, you might receive as few as 5 HTTP requests. Note that, by default, ICP queries are also written to the *access.log*. This may cause your logfile to grow to unmanageable proportions. You can disable ICP logging with the *log_icp_queries* option.

Sample configuration:

```
acl SIBS src cache.isp1.net cache.isp2.net cache.isp3.net
acl CGI urlpath_regex cgi \?
icp_access allow SIBS !CGI
http_access allow SIBS !CGI
#miss_access deny SIBS
...
...
icp_access deny all
http_access deny all
```

8.3 Running a Cluster of Caches

If you want to operate a cluster of caches such that any URL is only kept in one of the caches, you need to enable the *icp_hit_stale* option. Recall this causes Squid to return ICP_HIT for stale objects, which means you can not have true sibling relationships with other caches outside of your administrative control.

Each cache's configuration needs to specify all the other caches as siblings. For example, with a cluster of three, the first would include:

```
cache_host cache2.mydomain.net 3128 3130
cache_host cache3.mydomain.net 3128 3130
icp_hit_stale on
```

Again, multicast may be used here to reduce the number of messages sent out.

Note that you have to be careful with the *visible_hostname* option in a cache cluster. You might want the *visible_hostname* to be the same for all caches, perhaps to hide the details of your cluster configuration. However, this will confuse Squid's loop-detection scheme. Squid inserts the *visible_hostname* into outgoing requests, so if one of your caches forwards a request to another, the second one will think it has detected a forwarding loop. The best way to alleviate this problem is to append a unique identifier to the *visible_hostname* in each configuration, such as:

```
visible_hostname cache.mydomain.net/1
```

8.4 Mutual Parents

A pair of caches can not specify each other as parents, unless they use the *cache_host_domain* restrictions. Without those restrictions, the hierarchy has a forwarding loop.

Consider an example where two organizations want to utilize each other's caches as parents, but only for servers local to their respective domains. The first cache can specify:

```
cache_host cache.a.net parent 3128 3130
cache_host_domain cache.a.net a.net
```

and the other can specify just the opposite:

```
cache_host cache.b.net parent 3128 3130
cache_host_domain cache.b.net b.net
```

9 Analyzing Squid's *access.log*

9.1 Cache Result

The cache result code appears in the fourth column of Squid's native *access.log* format. It indicates how the request was handled by Squid.

The following codes, beginning with **TCP_**, are for requests to the HTTP port (3128).

TCP_HIT

A valid copy of the requested object is in the cache. Squid does not forward the request.

TCP_MISS

The requested object is not in the cache.

TCP_REFRESH_HIT

The object is in the cache, but *stale*. Squid has forwarded an *If-Modified-Since* request and received a *Not Modified* response.

TCP_REF_FAIL_HIT

The object is in the cache, but *stale*. Squid has forwarded an *If-Modified-Since* request but it failed (e.g. connection timeout), so Squid sends the old (stale) object to the client.

TCP_REFRESH_MISS

The object is in the cache, but *stale*. Squid has forwarded an *If-Modified-Since* request and received a response with the new content.

TCP_CLIENT_REFRESH

The client issued a request with the *no-cache* pragma, so Squid forwards the request.

TCP_IMS_HIT

The client issued an *If-Modified-Since* request and the object is in the cache, and still *fresh*. Squid does not forward the request.

TCP_IMS_MISS

The client issued an *If-Modified-Since* request for a *stale* object. Squid forwards the request as a miss.

TCP_SWAPFAIL

The object is believed to be in the cache, but could not be accessed, so Squid forwards the request.

TCP_DENIED

Access is denied for this request.

The following codes, beginning with **UDP_**, are for requests to the ICP port (3130).

UDP_HIT

A *fresh* copy of the requested object is in the cache.

UDP_HIT_OBJ

Same as UDP_HIT, but the object data is small enough to also be sent in the UDP reply packet.

UDP_MISS

The requested object is either *stale* or not in the cache at all.

UDP_DENIED

Access is denied for this request.

UDP_INVALID

An invalid request is received.

UDP_MISS_NOFETCH

The replying cache recommends that this request NOT be made at this time. Squid uses this reply code when reloading metadata at startup, and when the failure ratio exceeds its threshold.

There are also numerous error codes, which begin with **ERR_**, for requests to the HTTP port. We will not discuss these codes here.

9.2 HTTP Status

Every HTTP response includes a status code on the first line. This three-digit numeric code also appears in the fourth column of Squid's native *access.log*. Section 6.1.1 of *the HTTP/1.1 RFC* `<ftp://ds.internic.net/rfc/rfc2068.txt>` includes a full list of these codes. We are primarily interested in only two of them.

200 OK

This, the most common status code, indicates a successful request.

304 Not Modified

This means the request included an *If-Modified-Since* header with a timestamp, and the resource has not been modified since the given time.

9.3 Hierarchy Code

The ninth column of Squid's native *access.log* is a code that indicates how the next-hop cache was selected.

DIRECT

Squid forwards the request directly to the origin server.

FIREWALL_IP_DIRECT

Squid forwards the request directly to the origin server because the origin server's IP address is inside your firewall.

FIRST_PARENT_MISS

Squid forwards the request to the parent cache with the fastest weighted round trip time.

FIRST_UP_PARENT

Squid forwards the request to the first available parent in your list.

LOCAL_IP_DIRECT

Squid forwards the request directly to the origin server because the origin server's IP address matched your *local_ip* list.

SIBLING_HIT

Squid forwards the request to a sibling cache that sent us a UDP_HIT reply.

NO_DIRECT_FAIL

Squid cannot forward the request because of firewall restrictions and no parent caches are available.

PARENT_HIT

Squid forwards the request to a parent cache that sent us a UDP_HIT reply.

SINGLE_PARENT

The request is forwarded to the only parent cache appropriate for this request. Also requires that *single_parent_bypass* be enabled.

SOURCE_FASTEST

Squid forwards the request to the origin server because its *source_ping* reply arrived first.

PARENT_UDP_HIT_OBJ

Squid received the object in a UDP_HIT_OBJ reply from a parent cache.

SIBLING_UDP_HIT_OBJ

Squid received the object in a UDP_HIT_OBJ reply from a sibling cache.

DEFAULT_PARENT

Squid forwarded the request to a *default* parent, without sending any ICP queries first.

ROUNDROBIN_PARENT

Squid forwarded the request to the *round-robin* parent with the lowest use count, without sending any ICP queries first.

CLOSEST_PARENT_MISS

Squid forwarded the request to the the parent whose ICP_MISS reply has the lowest measured RTT to the origin server. This only appears with *query_icmp* enabled in the configuration file.

CLOSEST_DIRECT

Squid forwarded the request directly to the origin server because Squid measured a lower RTT to the origin than any of its parent caches.

NONE

Squid does not forward the request at all.

Note, when the two second timeout occurs waiting for ICP replies, the word **TIMEOUT_** is prepended to the hierarchy code.

9.4 Hierarchy Categorization

We find it useful to categorize request forwarding through the hierarchy in one of five ways. For the NLANR caches, we examine the *access.log* fields for some of the codes mentioned in the previous sections. This analysis is somewhat specific to our cache configuration. For example, we do not use the *default* option, and therefore do not need to look for DEFAULT_PARENT requests. Much of our information comes from utilizing ICP.

1. **Local cache hits** are requests satisfied from the local cache, which in our analysis includes anything with 'HIT' in the cache result code, even TCP_REFRESH_HIT and TCP_IMS_HIT.
2. **Remote cache hits** are requests forwarded to a neighbour cache, and which are cache hits there. The only way we can know this statistic is through ICP usage, by counting the number of requests forwarded due to an ICP_HIT reply.
3. **Remote cache misses** are requests forwarded to a neighbour cache, and which are cache misses there. Again, we know this statistic through ICP, by counting the number of requests forwarded due to an ICP_MISS reply.
4. **Direct retrievals** are requests forwarded directly to origin servers. It is likely that such requests are either 1) non-hierarchical, 2) because no ICP replies were received, or 3) forwarded to an origin server that is closer to the local cache than any of the cache's neighbors are.
5. **Other** requests do not fall into the above categories, such as requests denied by the access controls.

We are also interested in the number of requests suffering the two second *neighbour_timeout*.

We use the following simple Perl script to parse Squid's native *access.log* and output the percentage of each category:

```
#!/usr/local/bin/perl
while (<>) {
        chop;
        @F = split;
        $L = $F[3];                     # local cache result code
        $H = $F[8];                     # hierarchy code
        next unless ($L =~ /TCP_/);     # skip UDP and errors
        $N++;
        $timeout++ if ($H =~ /TIMEOUT/);
        if ($L =~ /HIT/) {
                $local_hit++;
        } elsif ($H =~ /HIT/) {
                $remote_hit++;
        } elsif ($H =~ /MISS/) {
                $remote_miss++;
        } elsif ($H =~ /DIRECT/) {
                $direct++;
        } else {
                $other++;
        }
}
printf "TCP-REQUESTS %d\n", $N;
printf "TIMEOUT %f\n", 100*$timeout/$N;
```

```
printf "LOCAL-HIT %f\n", 100*$local_hit/$N;
printf "REMOTE-HIT %f\n", 100*$remote_hit/$N;
printf "REMOTE-MISS %f\n", 100*$remote_miss/$N;
printf "DIRECT %f\n", 100*$direct/$N;
printf "OTHER %f\n", 100*$other/$N;
```

This script may or may not be appropriate to your particular situation, but should help you get started developing one of your own.

9.5 The NLANR Caches

Next we present a case study analysis of the NLANR cache hierarchy. Presently there are six cache servers in our top-level configuration, distributed across the U.S., with two on the Eastern coast, two in the Central region, and two on the West coast. Five of these caches communicate with each other via the vBNS, an unsaturated nationwide OC3/OC12 network with roughly 40 millisecond round-trip times between sites. The other cache, 'SV', is located at FIX-West, a popular exchange point in Silicon Valley, California.

The five vBNS caches use multicast to send ICP queries to one another for almost every request. The 'SV' cache does not send ICP to the other five because of the congested, commodity Internet paths between them. However, 'SV' and all of the other caches do have parent relationships with caches located in other countries.

The NLANR caches are all configured as mutual parents. That is, every cache is a parent for every other. Normally this would result in forwarding loops, but we eliminate them with the *cache_host_acl* directive:

```
cache_host nlanr.mcast.ircache.net multicast 3128 3130 ttl=4
acl nlanr_peers src it.cache.nlanr.net
acl nlanr_peers src pb.cache.nlanr.net
acl nlanr_peers src uc.cache.nlanr.net
acl nlanr_peers src bo.cache.nlanr.net
acl nlanr_peers src sd.cache.nlanr.net
cache_host_acl nlanr.mcast.ircache.net !nlanr_peers
```

which lets us query the multicast group, unless the request is from one of our own caches.

Also, for the five vBNS caches, we use the *proxy-only* option to effect a large, distributed cache. We also use the ICMP measurements to route requests to the closest neighbour.

Note that one of the machines, 'SD', is currently out of operation due to hardware failure, so we are unable to present its statistics here.

9.5.1 Hierarchy Analysis By Category

Here we present graphs of the hierarchy category values for the NLANR caches, plotted over a 30-day time period. These are figures 2–6.

9.5.2 Hierarchy Analysis By Cache

In figures 7–11 we show the same information, except this time grouped by cache instead of by category. The categories are shown *cumulatively* in each graph: (1) *local hits*, (2) *remote hits*, (3) *remote misses*, (4) *direct retrievals*, and (5) *other*. For each graph, the final sum should lie at exactly 100%. Note that the plots for *direct* and *other* coincide exactly because the *other* category is very near zero.

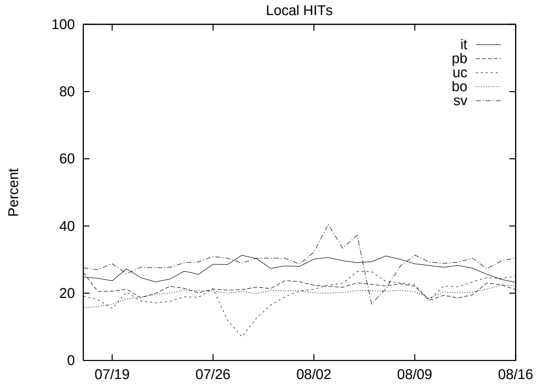

Figure 2: The percentage of local hits hovers in the 20-30% range. Note these are hits by request count, not by byte volume. The occasional dips are due to times when the entire cache was lost, or perhaps going through a slow reload after a crash. The interesting peak at 40% for 'SV' is due to an erroneous change in its configuration file. We inadvertently increased its cache size when we should not have.

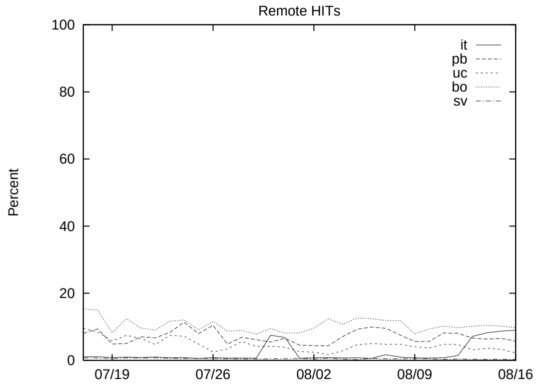

Figure 3: The percentage of remote hits is comparatively low, but not unexpectedly. The value normally ranges from 0-10%, but occasionally up to 15%. Notice that 'IT' is usually near zero, except for a couple of peaks. This is most likely due to multicast connectivity problems at that site. This looks like multicast connectivity was usually broken, but actually working for a small number of days. We will see this same pattern for 'IT' in almost all of the graphs.

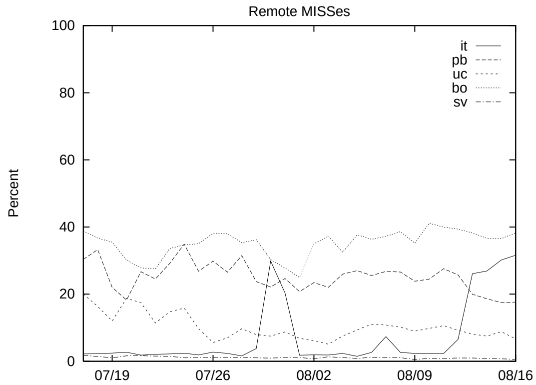

Figure 4: There is much more range in the percent of requests which are remote misses. Note that 'SV' is down in the 1% range because it does not peer with the other NLANR caches (but it does peer with other non-NLANR caches). The 'BO' cache usually forwards 35-40% of its requests to one of the other caches. Since the NLANR caches utilize the ICMP measurements, this likely means that 'BO' generally measures larger RTTs to the origin servers. At the other extreme, 'UC' forwards relatively few requests. Presumably, then, 'UC' is rather well-connected. Again notice the peaks for 'IT' due to changing multicast connectivity.

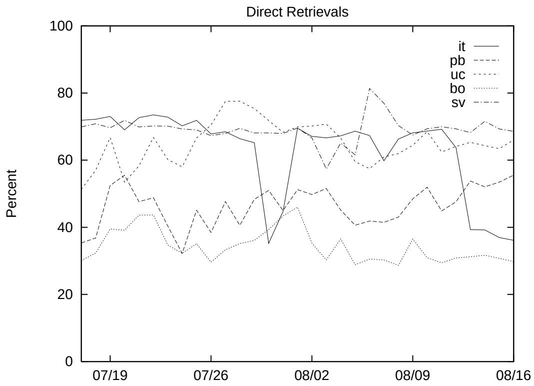

Figure 5: There is a lot of range in the percent of requests forwarded directly to origin servers. The 'SV' cache is near the top with 75%, as is 'IT' when it becomes cut off from the others. Whereas 'BO' is the highest for remote misses, it is the lowest for direct retrievals.

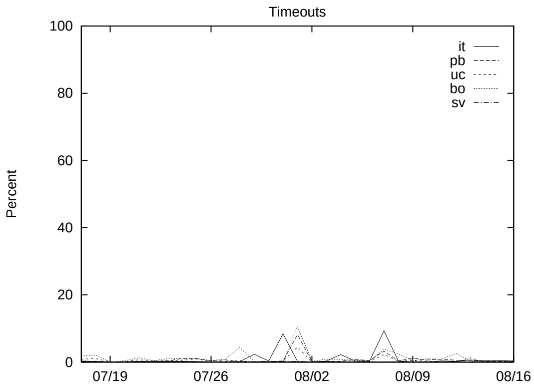

Figure 6: The percent of requests that timeout while waiting for ICP replies is very important to us because timeouts will adversely affect latency experienced by end users. Ideally we should see 0-1% timeouts. However, on a handful of days, it reaches as high as 10%. One interesting note is that even when 'IT' has lost multicast connectivity, it does not also have a large amount of timeouts, an indication that our technique for counting multicast peers works well. Indeed, most timeouts arise from *unstable* network conditions (either multicast or unicast). If the network is stable, then Squid will adapt to its neighbours being up or down. However, if things are unstable, then Squid too will be unstable and not able to accurately predict the number of ICP replies it should expect.

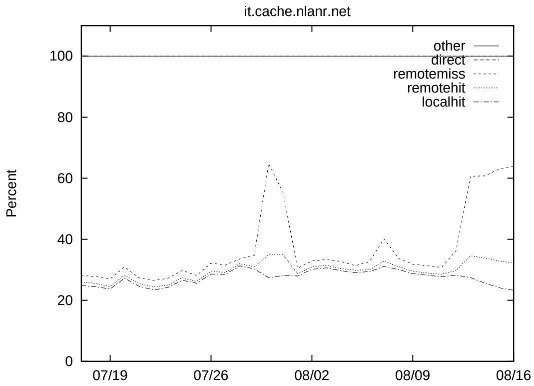

Figure 7: This figure indicates strong spikes when 'IT' is able to reach its neighbours. At the other times, we can see there are relatively few remote hit and remote miss requests; the few present are likely due to non-NLANR peer caches.

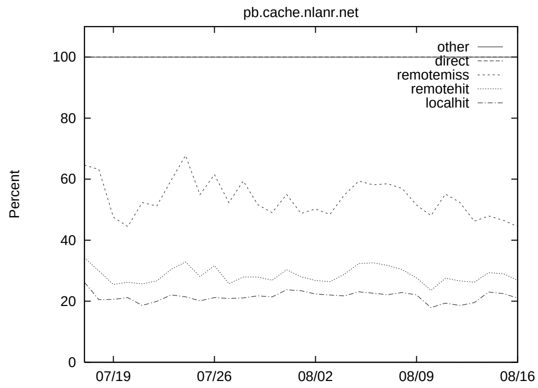

Figure 8: 'PB' shows a nice distribution of hierarchy categories. Approximately half of all the requests 'PB' forwards directly.

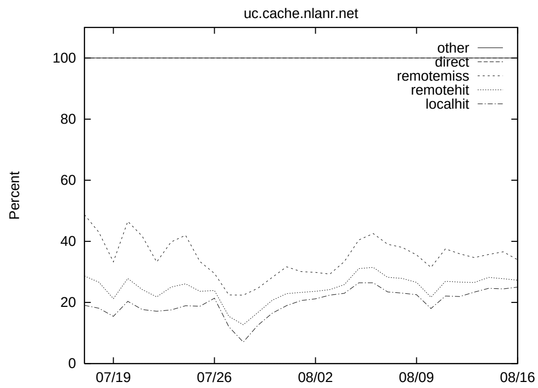

Figure 9: Here again it is apparent that 'UC' forwards many requests directly to origin servers, and relatively few to other caches. The dip near the end of July is due a temporary disk failure which caused the entire cache contents to be lost.

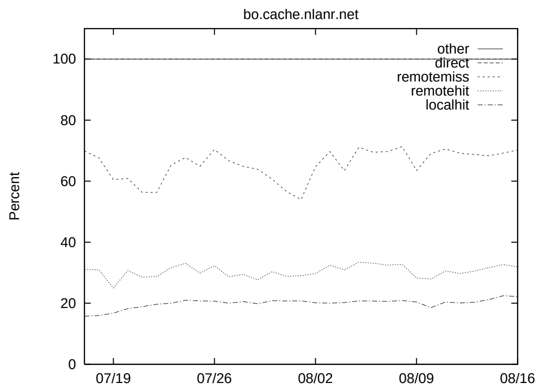

Figure 10: The 'BO' cache also shows a nice distribution of categories. Because this cache is less-well-connected than some of the others, it ends up forwarding many requests to its neighbours.

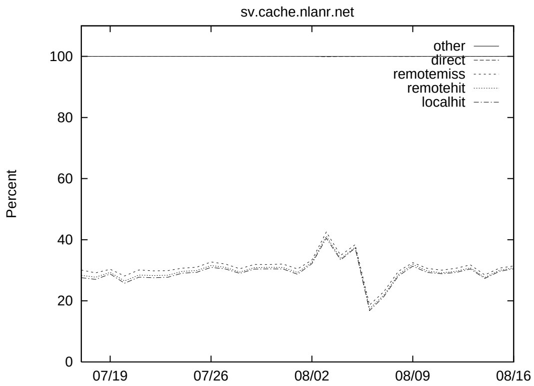

Figure 11: This graph clearly shows that 'SV' handles almost all of its requests as cache hits, or forwards directly to origin servers. Recall that 'SV' does not send ICP queries to the other NLANR caches. The increase in cache hits in early August was due to a misconfiguration error. The subsequent dip is due to a slow reload.

During this 30-day period, three of the caches were upgraded from 256 to 512 MB of RAM, and from 8 to 16 Gbytes of disk allocated for caching. Can you spot any corresponding changes in these graphs? Although difficult to detect, there are slight increases (roughly 5%) in the local hit rates for the three caches which were upgraded ('IT', 'UC', and 'BO'). This highlights another interesting thing about caching; doubling your capacity of a given resource does not necessarily double your hit rates!

9.6 Checking on your Siblings

If you are a sibling for another cache, you can use a simple command to search for anomalous requests. You should expect that a request from a sibling would result in a cache hit for you. If the IP address of the other cache is 10.0.0.1, then this command will locate any questionable requests:

```
awk '$2 == "10.0.0.1" && $4 ~ /MISS/' access.log
```

A few such requests likely does not merit concern; ICP and HTTP do not match up perfectly in all situations, so some of these requests may slip by. However, if you see a large number of anomalous *access.log* entries, the other cache could be using you as a parent instead of a sibling.

10 The Cache Manager

Squid's cache manager interface can provide you with some helpful statistics as well, which you can access either with the *cachemgr.cgi* program and a browser, or with the *client* program on the command line.

10.1 Server List

The *server list* (a slight misnomer) will report statistics about your neighbour caches. To view this information from the command line, use:

```
client cache_object://localhost/server_list | less
```

A subset of sample output is shown below.

```
Parent     : uc.cache.nlanr.net/3128/3130
Status     : Up
AVG RTT    : 31 msec
ACK DEFICIT:        0
PINGS SENT :        0
PINGS ACKED:    36133   0%
FETCHES    :     9765  27%
IGNORED    :     9193  25%
Histogram of PINGS ACKED:
        ICP_HIT :     2421   7%
        ICP_MISS :    33712  93%
Last failed connect() at: 18/Aug/1997:01:50:56 -0700
DOMAIN LIST: com edu net org gov mil us

Parent     : sv.cache.nlanr.net/3128/3130
Status     : Up
AVG RTT    : 69 msec
ACK DEFICIT:        0
PINGS SENT :     1007
PINGS ACKED:      964  96%
FETCHES    :      398  41%
IGNORED    :      322  33%
Histogram of PINGS ACKED:
        ICP_HIT :       68   7%
        ICP_MISS :      896  93%
DOMAIN LIST: au nz aq jp kr sg vn th tw lk
```

Squid will let you know if it thinks the neighbour is currently *up* or *down*. If a TCP connection attempt fails, Squid shows the time of the most recent failure near the bottom of this display. An important value to watch is the *Average RTT*, which is not simply a ping-based network RTT, but rather the average time taken to send an ICP query and receive an ICP reply. In other words, this value includes the time required by the peer to process the ICP message.

Ack Deficit is the number of consecutive unacknowledged ICP queries. If this number reaches 20, Squid will mark the peer as *down*. *Pings Sent* is the number of ICP queries sent to the peer, and *Pings Acked* is the number of replies received. Note that we are using multicast here, and *Pings Sent* only counts queries sent directly (via unicast) to the peer. *Fetches* is the number of HTTP requests forwarded to that neighbour. *Ignored* is the number of ICP replies that Squid has ignored, most likely because they arrived after Squid had already forwarded the request.

The histogram depicts the breakdown of the (non-ignored) ICP replies, with the percent based on the *Pings Acked* value, and not on the *Pings Sent* value.

Finally we display the domain list for each peer, as specified with the *cache_host_domain* directives. Note that we do not display the *cache_host_acl* configuration.

10.2 Client List

The *client list* tells you about your cache clients, including other caches using yours as a parent or sibling, although they are not identified as such because there is no easy way to identify that the request came from a browser versus another cache. This information is particularly useful if you have disabled ICP query logging. To view the client list, select the "Client List" menu option in the CGI program, or issue the following Unix command:

```
client cache_object://localhost/client_list | less
```

Some sample output is shown below. For each client IP address, Squid counts the number of requests for each cache result code, and displays their percentages.

```
Address: 205.17.46.36
Name: 205.17.46.36
ICP Requests 293995
        UDP_HIT                  11879   4%
        UDP_HIT_OBJ              14513   5%
        UDP_MISS                267515  91%
        UDP_INVALID                 39   0%
        UDP_MISS_NOFETCH            49   0%
HTTP Requests 40068
        TCP_HIT                    798   2%
        TCP_MISS                 30904  77%
        TCP_REFRESH_HIT           1894   5%
        TCP_REF_FAIL_HIT            36   0%
        TCP_REFRESH_MISS          1646   4%
        TCP_CLIENT_REFRESH        3116   8%
        TCP_IMS_HIT                818   2%
        ERR_READ_TIMEOUT            16   0%
        ERR_READ_ERROR              35   0%
        ERR_CLIENT_ABORT           257   1%
        ERR_CONNECT_FAIL           425   1%
        ERR_DNS_FAIL               106   0%
        ERR_ZERO_SIZE_OBJECT        17   0%

Address: 195.30.27.114
Name: 195.30.27.114
HTTP Requests 1855
        TCP_HIT                    209  11%
        TCP_MISS                  1295  70%
        TCP_REFRESH_HIT             95   5%
        TCP_REFRESH_MISS            90   5%
        TCP_CLIENT_REFRESH         106   6%
        TCP_IMS_HIT                 35   2%
        ERR_CLIENT_ABORT            22   1%
        ERR_CONNECT_FAIL             2   0%
        ERR_DNS_FAIL                 1   0%
```

The above numbers are real, but the IP addresses have been changed. The first client is obviously a peer cache because it uses ICP. Note that 9% of the ICP replies are HITs; approximately the same percent of *remote hits* in the previous section.

The second client does not use ICP, but that doesn't necessarily mean it is not a cache. It might just be configured to use the *no-query* and *default* options.

The *Name* field will most often just show the IP address, unless Squid has been configured to lookup fully-qualified domain names with the *log_fqdn* option.

10.3 Network Probe Database

Normally you should not have much reason to examine the network probe database (aka ICMP measurements), other than to satisfy your own curiosity. It is actually quite interesting to look at the data. Select "Network Probe Database" from the *cachemgr.cgi* menu, or issue the following command from a Unix shell:

```
client cache_object://localhost/stats/netdb | less
```

The output should resemble:

```
Network          recv/sent     RTT  Hops Hostnames
198.62.160.0        2/   2    22.0   7.0 www.jec.edu www.countrystars.com
    uc.cache.nlanr.net        82.0   9.0
    pb.cache.nlanr.net        84.0   9.0
    sd.cache.nlanr.net        85.0  16.0
128.2.194.0         2/   2    41.5   6.0 tom.cs.cmu.edu robotweb.ri.cmu.edu
    pb.cache.nlanr.net         7.0   5.0
203.183.232.0       1/  69   229.9   8.5 bbw.tokyoweb.or.jp powerhouse.co.jp
    sv.cache.nlanr.net       184.0   8.0
194.77.86.0         9/  23   466.9   9.6 www.ghg-verlag.com www.dom.de
    pb.cache.nlanr.net       283.0  12.0
    it.cache.nlanr.net       366.0  12.0
    sd.cache.nlanr.net       388.0  11.0
    uc.cache.nlanr.net       419.0  13.0
```

First of all note that hostnames are aggregated by "/24" networks. We assume that all hosts whose IP addresses have the same first three octets will have similar network measurements. In the example above we have only shown two hostnames per network, but in practice you will often see tens or hundreds of names per entry.

The lines starting with IP networks hold measurement data from your local cache. We also count the number of pings sent and received, although this information is not currently used in the neighbour selection process. We calculate decaying averages for the measured round trip time, and the estimated hop count. The RTT measurements are straightforward. Squid estimates hop counts by looking at the `ip->ttl` field of the ICMP reply packet, and guessing at its initial value when sent from the originating host. Because the hop count is averaged over time, and for different hosts, we may end up with non-integer values. Recall that we only use the RTT values in determining proximity, and not the hop counts.

The indented lines represent measurements returned by your neighbour caches in their ICP replies. These are listed in order of increasing RTT. Note that we won't necessarily have measurements for every peer cache. The measurements may time out, or may have failed to arrive.

The data above was taken from *bo.cache.nlanr.net*. For the first entry (e.g. www.countrystars.com) we can see that 'BO' is closer than any of the listed neighbours. Even so, we will still use ICP for a subsequent request. However, unless one of the neighbours returns a RTT measurement lower than 22 milliseconds, 'BO' would forward the request directly to the origin server.

The second entry shows that 'PB' is closer than 'BO' for some servers in the `cmu.edu` domain. Similarly, the third entry shows that 'SV' is closer for some servers located in Japan, reasonable since traffic from 'BO' to Japan likely routes through FIX-West anyway.

The final entry is interesting because 'BO' has the lowest estimated hop count, but the largest measured RTT. Because we only use RTT to select, a request to one of those domains would very likely be forwarded to 'PB'.

By default Squid keeps track of approximately 1000 networks in this database. For large caches, this value is probably too small, and you can configure its range yourself with the *netdb_low* and *netdb_high* configuration directives. The NLANR caches use values of 9900 and 10000 respectively. Also, the default *netdb_ping_period* is conservatively set at 5 minutes. You may want to increase the period to every one minute.

11 About This Document

This document was written in SGML, and then converted to other formats with the *SGML-Tools package* `<http://web.inter.NL.net/users/C.deGroot/sgmltools/>`. This paper is available in *HTML* `<tutorial.html>`, *letter-sized Postscript* `<tutorial.ps>`, and *A4-sized Postscript* `<tutorial-a4.ps>`.

11.1 Revision

`$Id: tutorial.sgml,v 1.41 1997/08/19 07:34:16 wessels Exp $`