DNS predictive caching: DNSCACHE

Greg Kochanski, Mike Andrews
{gpk,mikea}@bell-labs.com
Bell Labs, Murray Hill, NJ

June 18, 2001
(2001/06/18 03:47:15 UTC)
(v. 1.16)*

Abstract

The modern Internet has strong correlations between DNS requests: thus, network performance can be improved by predicting the future DNS requests. DNS requests can be a large or even the largest part of the time it takes to see a web page. At least one variant of this patent is a network element that Lucent could build. One variant is a HTTP web cache. Other variants could be licensed. Market: IP networking.

1 Introduction

As people get higher speed connections to the Internet, DNS lookup time becomes a larger fraction of the total. This is because DNS request and response packets are small, and little of the time is spent in data transfer to the home computer: most is spent in network delays or DNS server response time. Consequently, DNS lookup time is almost unchanged between a 14.4k modem and a 300k DSL (or faster) connection, and can become dominant under some conditions.

There are really two separable ideas here: DNS interception and predictive caching.

Note that the predictive cacheing idea also applies to the HTTP protocol, so one can have a predictive web cache that fetches pages before you ask for

*CVS/SSH repository accessible at `mikeaw2.div111.lucent.com`, user `patent`, module `pred`

them. This might well be the most commercially significant variant of the patent.

2 DNS interception

DNS interception: one can build a specialized router that identifies DNS packets, clients and servers via statistical or heuristic techniques. It transparently sends through all other packets, but acts on the DNS packets as a DNS server. The resulting system is transparent to all users, except for the time needed to get DNS responses. For almost all users in almost all circumstances, replies should come sooner, so overall network performance should be better.

The system could be made a function of a network element (firewall, bridge, router) by intercepting the easily recognized DNS requests. The system would check requests by port, and inspect their contents to identify true DNS servers. It can also can check contents of replies, to reduce the probability of false interceptions.

DNS interception introduces a new dns server in the data path. Under worst-case conditions, it can increase network delays, such as when a single person accesses a fast DNS server over a fast network. Under such a case, the extra computation time associated with DNS interception will slow things down. However, frequently requested addresses are likely to be in the DNS interceptor's cache, and can therefore be returned rapidly. If the primary servers of the frequently requested addresses are distributed across a continent, then the round-trip times would be long in the absence of DNS interception, and interception is likely to speed delivery of IP addresses. This is actually the case.

3 Predictive Cacheing

Predictive caching: Building up a statistical model of which IP addresses refer to which. Once that model is build, you can use it to anticipate the client's DNS requests, and fetch the information just before the client needs it. This correlation information will doubtless change as content changes, although perhaps not by alot.

Note that all the logic applies to HTTP requests.

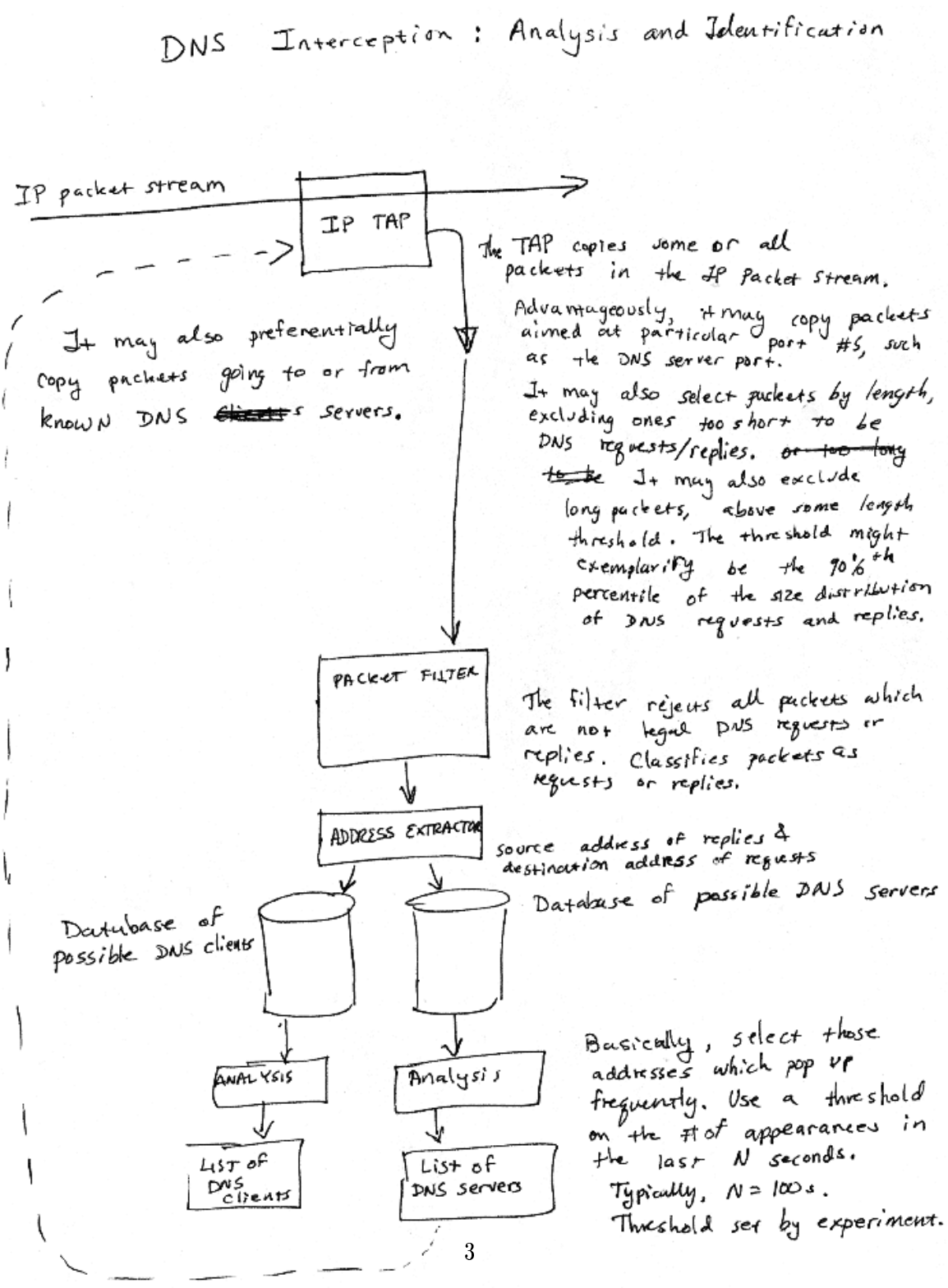

Figure 1: notes, page 1

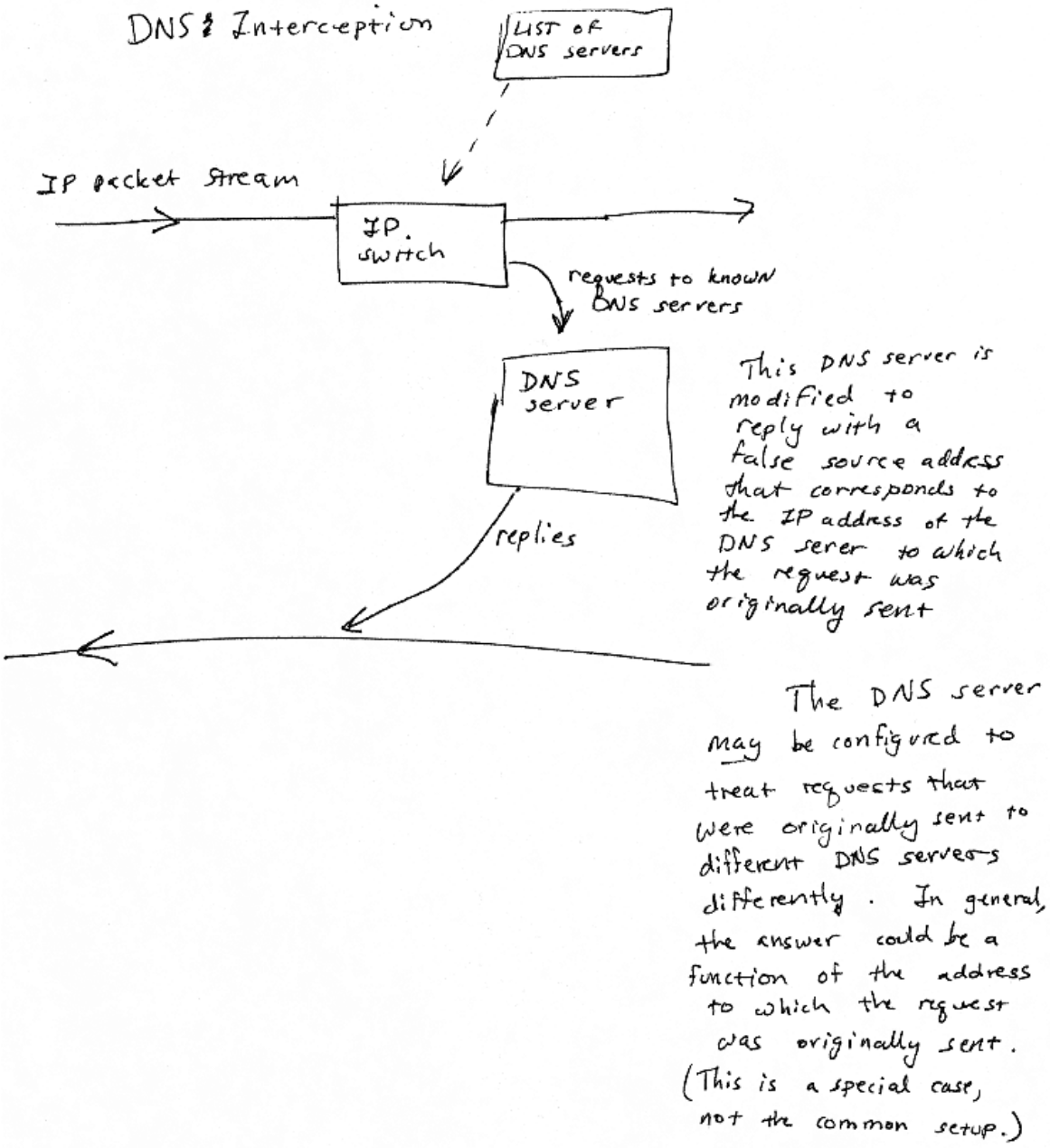

Figure 2: notes, page 2

DNS Interception : comments

The closest prior art is a firewall. If we want this idea to stand on its own, we have to distinguish from a firewall, (firewall that includes a DNS server).

Firewall

Set to operate on a specific port #.

Running a DNS server on a firewall requires computers inside the firewall to know the address of the firewall DNS server. This requires customers to hand-specify that address or to use DHCP servers associated with the firewall.

Clients need to know, and need to change their computer's configuration.

No analysis is performed to identify DNS servers.

DNS Interception

Can detect DNS traffic on nonstandard port #s.

Computers behind the DNS interceptor can be configured with any DNS server's address. Simpler configuration & administration, especially if we're talking about an ISP (where each computer has a different owner, and there is no central organization to enforce policies.)

The clients don't need to know that a DNS interceptor has been added. They will see no change.

From a patent perspective, the difference is that the interceptor analyzes the packet stream.

Figure 3: notes, page 3

3.1 General Discussion

Predictive caching can be implemented without DNS interception, but that option is less in line of Lucent's business.

Once you have DNS packets, you can do lots. Your servers can do predictive caching. For instance, web pages introduce long-range correlations between DNS requests, because once you have jumped to a web page, you are very likely to leave via one of its links. Each web server points to a set of other servers, and that set is often small. Thus, when you receive one DNS request, you have a shot at predicting the next. Many modern web pages refer to images on other servers (e.g., Akamai or banner ads), so these DNS lookups can be predicted with high probability.

this prediction will be even more effective because the user will be implicitly loading information from those other servers as his browser assembles a single web page. so it will certainly pay to load up all the dns requests on a particular page at the outset, even if the user has not asked for those pages yet. this is the mildest form of prediction because we know that in the course of assembling the page the user will have to ask for those other dns's.

It is probably best to predict future lookups as a function of a particular client machine's requests. First order statistics is probably appropriate (i.e. when making the lookup prediction consider one past DNS request at a time). We assume that each past DNS request corresponds to a web page lookup on the referenced machine. We then calculate (from our database of previous DNS requests), the probability that the machine that made the first request will access other machines in the immediate future (several tens of seconds). We then look up the most probable M of those machines ($M = 5$ perhaps), either in our own cache, or by making external requests to other web servers.

Is efficient prediction possible? Yes. Few web servers refer to more than 100 other servers,[1] and has a perplexity[2] probably about 10, as some outgoing links are far more likely than others. That means the predictive server needs to know 10 addresses in order to anticipate the client's next request. Since typical lifetimes of DNS packets are 1000 seconds, most of those 10 needed addresses will be in the predictive server's cache so long as DNS requests for a given machine come in at the rate of 1 per 100 seconds

[1] do we have real statistics on this? GPK: Not formal, but I did enough sampling to justify "few".

[2] Perplexity is exp(entropy), and is the average number of choices you have a given point on a graph. This is just saying that on average, you have about 10 outgoing links from each web page.

or higher. At lower request rates, the predictive server will have to make multiple DNS requests per client DNS request.

In fact, the above estimate is conservative, because the rate calculation is over links, not over web servers, and many different web servers (or links?) will often link to the same point. The predictive DNS server will be able to efficiently maintain these common destinations in its cache even at low traffic rates.

Another optimization is to model the state of the client's DNS cache. One can (and should) be conservative by assuming the client has no other source of information. Then, since the DNS packets you send to the client have expiration times, you can know when previous information is invalid and in need of a refresh. If you know that the client's cache has some fresh entries, you can reduce the difficulty of predicting the next DNS request because you can assume that the client won't look up addresses that are already in its cache.

How does this interact with load balancing schemes or Akamai? A conservative approach (especially nice when both ideas are combined) is to simply stop intercepting DNS traffic for servers with short-lived data. The system described here doesn't interfere with Akamai's network mapping, because it preserves packet routes.

An extension is to watch all packets (in the packet interception variant), and predict DNS requests based on all packets instead only of prior DNS requests. This is more CPU-intensive, but may allow faster service. As an extreme, one could look inside packets for URL's, and predict DNS requests as a function of individual web pages.

This does not conflict with RFC2535 [1], as the server does not modify the information as it passes through, and acts in all respects (other than request timing) like a normal DNS server.

3.2 Collection of Other Information

Andy Pargellis suggests that you could collect other information, such as the mean time between DNS lookups, either as marketing information, or to improve DNS performance. Knowledge of the time, $T(X, Y)$, between looking up addresses X and Y will allow the DNS server to optimize its operation. In case of heavy loading, the DNS server can sort the requests that it sends out in order of when the resulting information is expected to be needed. Also, if $T(X, Y)$ becomes comparable to the lifetime of information in the DNS caches, the requests should be delayed so that the information doesn't time

out before the user requests it. Further, if $T(X, Y)$ is substantially larger than the lifetime of Y in the cache, it becomes improbable that the address of Y, from any single lookup, will still be valid when the user finally requests it. Thus, the system would either look up Y repeatedly to keep the cache fresh (if the system load is sufficiently light), or simply not bother looking up Y until it is requested (if system loads are heavy).

3.3 Cache Triage

The simplest form of prediction is to simply look at the average time between requests for a specific name to IP address translation (DNS request). One can then pick a threshold, and ensure that names requested more often than the threshold are always fresh in the cache. As each such frequently requested name approaches the end of its valid lifetime in the cache, the DNS server would look it up again.

The reason that this form of prediction helps is that mappings that are accessed about once per lifetime in the cache are frequently accessed *after* they have expired in the cache, because the accesses are random and unpredictable.

A programmer skilled in the art could design such a system. A simple implementation would just iterate (preferably at low priority) through the contents of the cache, and when it found an name/IP mapping that was (a) used often enough, and (b) likely to expire before the next time it would iterate through, the software would send out a DNS request to refresh said name/IP mapping. A more complex iteration might keep a list of pointers to name/IP mappings, kept sorted by expiration time. That way, the software need not continually iterate through the entire list of mappings.

The threshold for the average time between requests is F times the mapping's lifetime, where typically F is between 0.1 and 100, preferably with F between 0.3 and 10. The detailed value of F is best set by experiment, considering the network load, cache size, and available CPU cycles of the DNS server. Increasing F will increase all three factors, roughly proportionally (for $F > 1$).

Advantageously, the RMS time between requests can be used instead of the average time between requests. If all mappings are cached, using a median will result in more efficient operation if requests for a particular mapping come in bursts. Consider a short burst far away from other bursts, where the burst length is short compared to the lifetime of the mappings, and any mappings cached from the previous burst have expired. The normal

cache will handle all but the first request in the burst, so only the first request needs attention from a predictive cache. One needs, then, a measure of rate that corresponds more closely to the inter-burst spacing than does the average time between requests.

We also consider other algorithms for measuring burst spacing. The simplest is to simulate the aging process in an assumed infinite cache: the requests for a particular mapping are time-stamped and marked with their lifetime. The computer then iterates (in the order in which they were recieved) over said list of requests, and marks the ones which are within a lifetime of an unmarked request. The marked requests correspond to those that could be serviced from a cache, and the unmarked one correspond to requests that would have to be sent out over the network. The result of the burst-spacing algorithm is then the average (or median) time between unmarked requests.

3.4 Prediction from DNS requests

This is written in the context of a server on a firewall, but it isn't specific to firewalls. There are four processes running simultaneously. One collects statistics, one maintains that statistical tables, one does DNS lookups in the background at low priority, to fill up the cache, and one collects DNS responses from outside servers. There is also a cache of DNS information (known mappings from names to IP addresses that have not yet expired), and one or more tables of access statistics. The tables contain information on how often a request for name X is followed (within reasonable proximity, defined below) by a request for name Y.

3.4.1 Statistics process

1. Receive DNS request from client C.
2. Look up in cache, reply if possible.
3. If not, pass C's request on to an appropriate "outside" server.
4. For each table of statistics, do the following:

   The following is a conceptual description, not a practical implementation, though one skilled in the art could write a practical program from it. The current request is denoted as request 0.

(a) For each of the M most recent previous requests from Client C, denote said previous request by k. If request k is no more than τ seconds old, add to (or increment, as appropriate) an item in the table that indicates that the machine name of request k is followed by a request for the machine name of request 0.

5. Repeat.

Experiment is the best means of finding M and τ, but in the preferred implementation, there are 0 to 10 different statistical tables, each constructed from different combinations of M in the range of 1 to 1000 and τ in the range 0.01 second to 1000 second. Preferably, one of the tables has $M = 1$ and $\tau > 100s$, and preferable another of the tables has $M > 3$ and $\tau < 1s$.

The above increment is normally the integer 1, but we consider the possibility that the increment is a function of k and age. In such a case, normally the function would decrease as either k and age increases.

3.4.2 Table Maintenance Process

Low priority.

1. Iterate through all the entries in all the statistical tables.
2. If entries are older than some threshold T_{max}, delete them.
3. If the table occupies too much memory, delete the oldest entry.
4. Repeat.

T_{max} is best set by experiment. It is the time over which the correlations between DNS requests changes. Typically, T_{max} is between 10 seconds and 1000 days, more typically between 1000 seconds and 100 days.

3.4.3 Alternative Table Maintenance Process

Table Maintenance can also be accomplished by a "leaky integrator" scheme.

Low priority.

1. Iterate through all the entries in all the statistical tables.

2. Multiply them all by a factor q, where $0 < q < 1$. Normally, $q = e^{-\Delta T/T_{leak}}$, where $e = 2.71828...$, ΔT is the time between successive sweeps of the alternative table maintenance process, and T_{leak} sets the time over which the entries disappear.

3. If entries are smaller than some threshold q_{min}, delete them.

4. Repeat.

Normally, q_{min} is set at Q times the average increment (typically 1, but see above). Q is typically between 0.1 and 1.0 . T_{leak} is best set by experiment. It is the time over which the correlations between DNS requests changes. Typically, T_{leak} is between 10 seconds and 1000 days, more typically between 1000 seconds and 100 days.

3.4.4 Response handler process

1. When a DNS reply packet arrives, check that it matches an outstanding request.

2. Send the appropriate reply to client C (the client that initiated the request).

3. Cancel the outstanding request

4. Add the information to the local cache.

5. Repeat.

3.4.5 Cache fill process

Low priority.

1. When a request is recieved for name x, iterate through all the entries where X (the earlier of the two requests) = x, in all the statistical tables.

2. In a given entry, if Y (the later of the two requests) is not in the cache, *and* the value of the entry exceeds a threshold r_{fill}, where r_{fill} is described below, *then* send out a DNS request for Y.

3. Repeat.

The threshold, r_{fill}, is set as a function of the (X,Y) entry, and (most generally), all of the (X,*) entries. A special case of particular interest is where r_{fill} is a function of the (X,Y) entry and the sum of the (X,*) entries.

In the preferred implementation, $r_{fill} = r_i \cdot \bar{I} + r_f \cdot \sum_{y \in Y} I_{(X,y)}$, where r_i is a constant, r_f is a constant, $I_{(X,y)}$ is the entry indexed by (X, y), and $\bar{I}$ is the average increment. Typically, r_i is set by experiment between 0 and 100, more typically between 0.3 and 10. Typically, r_f is set by experiment between 0.01 and 0.9, more typically between 0.03 and 0.6. In the experiments to set r_i and r_f, one must consider the network loading (more DNS requests will be made as either constant is made smaller), the cache size (more cache space will be required if either is made smaller, and it is undesireable for the cache to overflow), and the computational load. One must balance these disadvantages/limitations against the expectation that the probability of having a mapping in the cache when the Response Handler Process needs it.

We contemplate that the constants r_i and r_f may be dynamically adapted, depending on the network load, available spare CPU capacity, and available unused cache memory.

Advantageously, the statistical tables may also contain information on the time interval between requests X and Y. Then, rather than sending out a DNS request, above, the system would schedule a DNS request to be sent out at some future time. In the case where the mean time interval between X and Y is comparable to (or larger than) the lifetime of Y's mapping in the cache, delaying the DNS lookup is advantageous, as it reduces the probability that the mapping will expire before it is used. In such a case, the histogram of the time delay between X and Y would be estimated from the available information (perhaps by calculating a maximum-likleyhood fit of some reasonable analytic form to the data), and then choosing the delay that maximizes the probability that the mapping will be used before it expires.

3.5 Operation with a Web Proxy

Assume a firewall with a HTML proxy H and a DNS proxy D. Inside the firewall is a client C. Outside are a bunch of servers S_i.

1. C sends HTTP request to H.
2. H passes on HTTP request to a server (assuming the HTML wasn't in H's cache).

3. When reply comes in, H passes HTML back to C.

4. H then computes likely future DNS requests, and sends that list to D. This can be implemented by having H simply make DNS requests to D for all the addresses that are likely to be used in the future. If so, D does not need any special protocol, and can be just a standard DNS server.

5. D then sends out requests for IP addresses

6. When the replies come in, it caches the results.

7. Later, when C requests a different URL, D will probably already have the address in its cache.

3.6 Prediction from DNS requests and a Web proxy

. The two techniques can be advantageously combined. All one needs is one more process which parses web pages as they pass through the proxy, and increments the appropriate elements in a statistical tables.

3.7 Alternative Operation with a Web Proxy

MRA: perhaps we also want to consider an addition to the html standard which will offer the client's browser a list of ip addresses it probably will need based on the current page being viewed. this could be in the form of a special tag (*e.g.,* a specially formatted comment) which certain browsers will have the option of interpreting. this sounds pretty nice to me actually. to be specific, see the example in subsection 3.8.

Assume a firewall with a HTML proxy H. Inside the firewall is a client C. Outside are a bunch of servers S_i.

1. C sends HTTP request to H for a page P.

2. H passes on HTTP request to a server S (assuming the HTML wasn't in H's cache).

3. When reply comes in, H passes HTML back to C.

4. H then computes likely future DNS requests, and sends that looks them up. It builds a cache of name/IP address mappings.

5. Later, when someone requests P, H will insert any relevant name/IP mappings that it has in its cache.

6. The web browser on C stores those name/IP mappings for use when the user clicks on a hyperlink.

3.8 example html format with predictive DNS information

A special HTML tag would be defined (here shown as "`!%%%%`"), which would carry a name, an IP address, and an expiration time for that address. Additional information could also be carried, such as SOA information and all the other stuff in a standard DNS packet.

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<body>
mike andrews' provisional web page
<p>click <a href="http://www.google.com">here</a> to go to google.
<p>click <a href="http://www.google.com">here</a> to go to yahoo.
<!%%%% DNS www.google.com:216.239.35.100 2001-04-31 19:32:14 UT >
<!%%%% DNS www.yahoo.com:216.32.74.51 2001-04-31 19:22:10 UT >
</body>
</html>
```

3.9 Low Priority DNS Lookups

Advantageously, when H notifies D of likely future requests, these requests can be marked specially, so that D treats them at a lower priority. Since the likely future requests from H are not certain to be used, they should not be allowed to interfere with the servicing of immediate requests where an answer is required (such as requests from C).

This marking can be accomplished by sending the requests to a separate TCP or UDP port from the normal DNS requests. Alternatively, this lower priority can be marked with a specific OPCODE field in the DNS request header (chosen in the range 3–15 [2]). Using this form of mark, the DNS server need not reply to a request, thus saving computation in D and H and network bandwidth.

3.10 Avoiding Loops and Oscillations

Gordon Wilfong notes that if these servers are set up wrong, they could oscillate and generate packet floods. The conditions to avoid involve the following scenario

- Server A recieves a request for X.
- Server A predicts that as a result of X, Y will be needed.
- Server A then sends the request for Y to server B.
- Server B recieves the request for Y, and predicts that X will be needed.
- Server B sends a request back to A.

This is normally self-limiting, because after 1 loop, the addresses of X and Y will be in both caches, and further lookups will be unnecessary. However, if caches get flushed for some reason (*e.g.,* a DNS record has a very short lifetime, or the cache is just too small for the traffic), the loop can occur again. Its also possible that the "phase space" for such a loop will be small, since it is likely that there are many more servers than just A and B (?).

There are several things that can be done to prevent runaway situations (GPK):

- Mark packets with a special bit [see section 3.9] that indicates that this packet is a predicted lookup, and not to be used to predict further lookups.
- Refrain from doing predictive lookups when record lifetime is very short (*e.g.,* < 5 s)
- Refrain from doing predictive lookups when cache overflows cause records to be dropped before they expire.
- Use these servers only between customers and the main internet. This fits in particularly well with the DNS interception scheme. In the interception scheme, all requests come from the left, and all outgoing queries go to the left; therefore loops are impossible.

4 Predictive HTTP proxies

Essentially, using the techniques above, the proxy looks at the links on a web page as it fetches it, and then pre-fetches images and pages to which the first page refers. The only extra functionality needed is code to determine which pages may be safely pre-fetched. One skilled in the art could, by experiment, write a set of heuristic rules that would make the operation of the predictive HTTP proxy transparent to all (or almost all) web sites. Only HTTP GET operations are generally safe to pre-fetch. Additionally, one should not normally pre-fetch dynamically generated pages (which can sometimes be identified by a '.cgi' or '.asp' at the end of the URL). Images are almost always safe to pre-fetch, as are static HTML pages.

References

[1] D. Eastlake. Domain name system security extensions, 1999. (RFC2535).

[2] P. Mockapetris. Domain names — implementation and specification, 1987. (RFC1035).