*Proceedings of USITS' 99: The 2nd USENIX Symposium on Internet Technologies & Systems*

Boulder, Colorado, USA, October 11–14, 1999

# A DOCUMENT-BASED FRAMEWORK FOR INTERNET APPLICATION CONTROL

William LeFebvre and Ken Craig

# Rapid Reverse DNS Lookups for Web Servers

William LeFebvre
*Group Sys Consulting*
*Alpharetta, GA 30022*
**WNL@GroupSys.com**

Ken Craig
*CNN Internet Technologies*
*Atlanta, GA 30348*
**Ken.Craig@CNN.com**

## Abstract

When a web server wants to learn the domain name of one of its clients, it must perform a lookup in the Domain Name System's "reverse domain", *in-addr.arpa*. These lookups can take time and may have an adverse impact on the web server's response to its clients. Rapid DNS is an intermediate client/server system that operates between a web server and a DNS server. It provides caching of the results and, more importantly, limits web server lookups to the data contained in the cache. This provides a significant improvement in response time for situations in which knowledge of the hostname is not critical to the web server's operation. The Rapid DNS system was implemented for use in the web farm that serves the collection of Cable News Network (CNN) sites. Its design is presented, along with measurements of its performance in the CNN environment.

## 1 Introduction

When a client connects to a server, the only information about the client that is available to the server is the client's IP address. In order to learn more about the client, the server must perform a DNS lookup in the *in-addr.arpa* domain, called a *reverse lookup*, to translate a client's IP address in to a name [9]. On widely accessed web servers, a high percentage of the reverse lookups will involve name servers from distant networks. Consequently, these lookups can take a long time.

Most high traffic web sites cannot afford to wait for the completion of reverse lookups, as the delay in processing these lookups would have a detrimental impact on the site's response time. Therefore, client tracking is limited to just IP addresses. Any desired demographic information must be generated off-line. Real-time determination of a visitor's origins is not a reasonable possibility due to the time required to perform a reverse lookup.

The CNN web farm supports approximately 50 web servers which provide content for sites known as *cnn.com*, *cnnfn.com*, *cnnsi.com*, and many others. A single web server in this farm can see as many as 20,000 hits per minute. The farm was designed from the beginning for simplicity, reliability and speed in order to support a web site that is the most heavily trafficked news site on the internet. In addition to serving over 20 million page views daily, the web farm must be able to withstand traffic spikes that are three times what is experienced on a normal day.

The farm consists of smaller, distributed servers which can be easily replaced or re-purposed. The web server software is primarily off-the-shelf, and additional software, in the form of web-server plug-ins, must not introduce significant latency to routine requests. Specialized functions are generally distributed off of the main-line servers to protect the basic service. A relatively homogenous environment simplifies the process of re-purposing hardware when the need arises.

Off-line DNS processing has provided the web farm team with useful information for analysis, but offers little benefit to advertisers; an important consideration for an advertising-supported web site. Domain based ad targeting was one of the most highly requested features to be added to our advertising capabilities due to its supposed simplicity and universal acceptance of accuracy. While architecturally simple, implementation of such a capability at scale requires a different solution.

As beneficial as such targeting may be, protecting the reliability of the primary web serving functions always takes precedence. This is basis for the two primary design requirements of any additions to the CNN Web Farm, including Rapid DNS; high performance during normal operations and graceful degradation of service under excessive traffic loads.

Even though it may be possible for a well-configured name server to handle several hundred requests per sec-

ond, there will still be a problem with latency. The root name servers are expected to sustain a minimum response rate of 1,200 queries per second, but they also disable recursion on all requests [7]. Server load may be an issue, but far more critical is the need to provide a quick response. A reverse lookup will need to consult name servers throughout the world and can take several seconds to complete. A server that receives over 300 requests per second cannot afford to have each request delayed by a recursive lookup.

In section 2 we present related work. Section 3 lays the foundational premise on which the entire system is built. Section 4 describes details of the Rapid DNS client, server, and protocol. Section 5 discusses management of the cache used by the server. Section 6 explains the specialized way in which IP addresses are queued internally for processing by DNS. Section 7 discusses the use of negative caching in Rapid DNS. Section 8 presents performance results for a variety of configurations. Section 9 discusses the results, and section 10 looks to the future.

## 2 Related Work

Surprisingly little work has been done in this area. A search of the published body of work has revealed no documented efforts to provide rapid reverse lookups for web servers.

Work has been done to utilize DNS for load balancing requests across multiple web servers ([1], [6]) and for integrating DNS lookups with HTTP redirection to achieve load balancing [3]. Brisco discussed the viability of using DNS as a general load balancing tool [2] and Schemers developed a Perl tool for tailoring DNS answers based on measured load [11]. A study of name server traffic on the NSFNet was conducted by Danzig, Obracza and Kumar [4], and they observed (among other things) that negative caching of DNS responses by servers would have little impact on the reduction of DNS packets across a wide-area network.

## 3 Premise: “I don’t know” is acceptable

The desire for instant domain name information on the CNN Web farm drove us to implement a mechanism for rapid resolution. The foundational philosophy for Rapid DNS is that the answer “I don't know” is acceptable. If a name is not readily available when requested, then Rapid DNS is free to answer “I don't know”. The web server then proceeds as if it never performed the lookup. This philosophy allows the implementation to uncouple queries from the actual resolution of the name

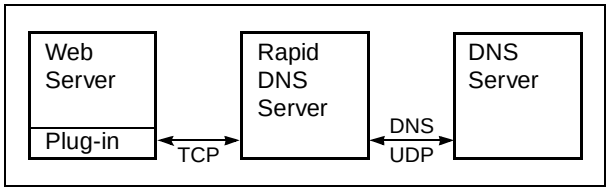


**Figure 1: Inter-Server Relationship**

Is it reasonable to accept non-answers for this sort of query? For our purposes the answer is yes. The host name information is needed for two separate purposes. First, we want to be able to produce summary traffic information correlated by top level domain. This allows us to calculate zone demographic information on our audience: “35% of our traffic was from educational sites” for example. Before the deployment of Rapid DNS we performed name resolution off-line: traffic logs carried the IP address and the translation to name was done *en masse* overnight after the logs were extracted from the web server. In this situation an address that cannot be translated in to a name is just placed in an “unknown” category. Clearly the same can be done with “I don’t know” answers. Although this will adversely affect the demographic results, it will not hamper the operation of the web server itself.

The second purpose for domain names is to drive the selection of advertisements presented on the page. If the web server knows the domain name of the client it has the option to choose an advertisement specifically targeted for an audience group implied by the domain. If the name information is missing then the advertisement selection can just draw from a generic pool of advertisements. We miss a chance to target an ad, but are still able to operate.

There is a situation in which the absence of a domain name will have an impact on the server’s operation: name-based authentication. If the server determines accessibility based on the domain name then the answer “I don't know” is not acceptable. A user who has legitimate access may not receive that access if the name lookup service can't provide an answer. The CNN web servers do not use name-based authentication, so this was not a concern for our service.

## 4 Design of the Rapid DNS

Rapid DNS is implemented as an intermediate service placed between the web server and the DNS name server (see Figure 1). A plug-in or module in the web server acts as the Rapid DNS client and is invoked as part of normal page handling. The Rapid DNS server can be run on any host accessible to the web server via

TCP, reading and responding to client requests. The Rapid DNS server is the only component that actually issues DNS queries. For the remainder of this paper, unless otherwise indicated, we will use the term "server" to refer to a Rapid DNS server and "client" to refer to a Rapid DNS client, even though that client may be part of a web server.

## 4.1 Client

The client we have developed for the Netscape Enterprise server uses a single persistent Rapid DNS connection to handle all requests in a given process, even though that process may have many threads handling HTTP requests

The design relies heavily on the multi-threaded capabilities of the Netscape server provided through its application programming interface (API) [10]. At initialization time, the Rapid DNS client starts several background threads. One thread, the writer, dispatches requests to servers. Additional threads, the readers, are created to read and process the servers' responses (one thread per server).

The Netscape server creates threads for handling HTTP requests. When a request arrives it is dispatched to an idle thread. While processing the HTTP request, the Rapid DNS plug-in function will be invoked and will place the peer's IP address on a central queue for processing. The writer thread takes a request off the queue, dispatches it and moves it to a pending queue that is specific to the server used. As a reader thread processes a response, it is matched up with the corresponding request in the pending queue. One response may serve to fill more than one request.

Request processing in the Netscape server is performed in 6 phases: authorization, name translation, path checking and modification, object typing, request servicing, and logging. Server plug-in functions can be invoked during any phase. Information is passed in to the functions using parameter blocks, or *pblocks*. These are hash tables that store name/value pairs and provide for easy lookup and modification. All information about the request is stored in a set of pblocks, and plug-in functions affect processing of the request by modifying these pblocks.

The Netscape API provides a function, *session_maxdns*, that retrieves the domain name of the HTTP client host. This function also stores the information in a pblock: specifically in the session client pblock using the parameter name *dns*. Subsequent calls to *session_maxdns* will use the information found there rather than perform the DNS lookup again. If DNS lookups are turned off in the Netscape server's configuration, then *session_maxdns* will not send any DNS queries, but it will still look in the pblock for the name.

The Rapid DNS plug-in, *rdns_lookup*, takes advantage of this behavior. When an answer arrives from the server, *rdns_lookup* will place it in the client pblock as the parameter *dns*. Subsequent calls to *session_maxdns* will never use DNS directly but rely exclusively on the information in the pblock, even if DNS lookups are turned off in the Netscape configuration. If the answer received from Rapid DNS is "I don't know" then no information is placed in the pblock, and calls to *session_maxdns* will return NULL. Beyond the change to the pblock, *rdns_lookup* affects no aspect of processing a request. As a consequence, it can be used in any phase of request processing. The following C statement illustrates how the name information is placed in the pblock:

```
pblock_nvinsert("dns", name, sn->client);
```

The *rdns_lookup* function must be invoked before any plug-ins that may need to utilize its information, such as advertisement scheduling software. CNN chose to use a configuration that invokes the client as the very last object type function, so that it is run immediately before entering the request servicing phase.

The writer thread will always dispatch requests to the server with the shortest pending queue. This policy automatically compensates for malfunctioning or abnormally slow servers. If a server fails, its persistent connection is severed and the client code will stop dispatching requests to the server. Finally, a watcher thread monitors all the queues to ensure that none of the requests get stuck.

This client design provides good scalability for web servers. In fact, we have Rapid DNS deployed across more than 60 web servers, and they all use the same trio of servers.

## 4.2 Server

The server also uses a threaded design. A thread is created to handle each client connection, and other service threads handle various maintenance tasks and communications with DNS servers. The server is logically separated in to two components: the front end and the back end. The front end handles the task of providing answers to clients, while communications with DNS servers is completely isolated to the back end. This total decou-

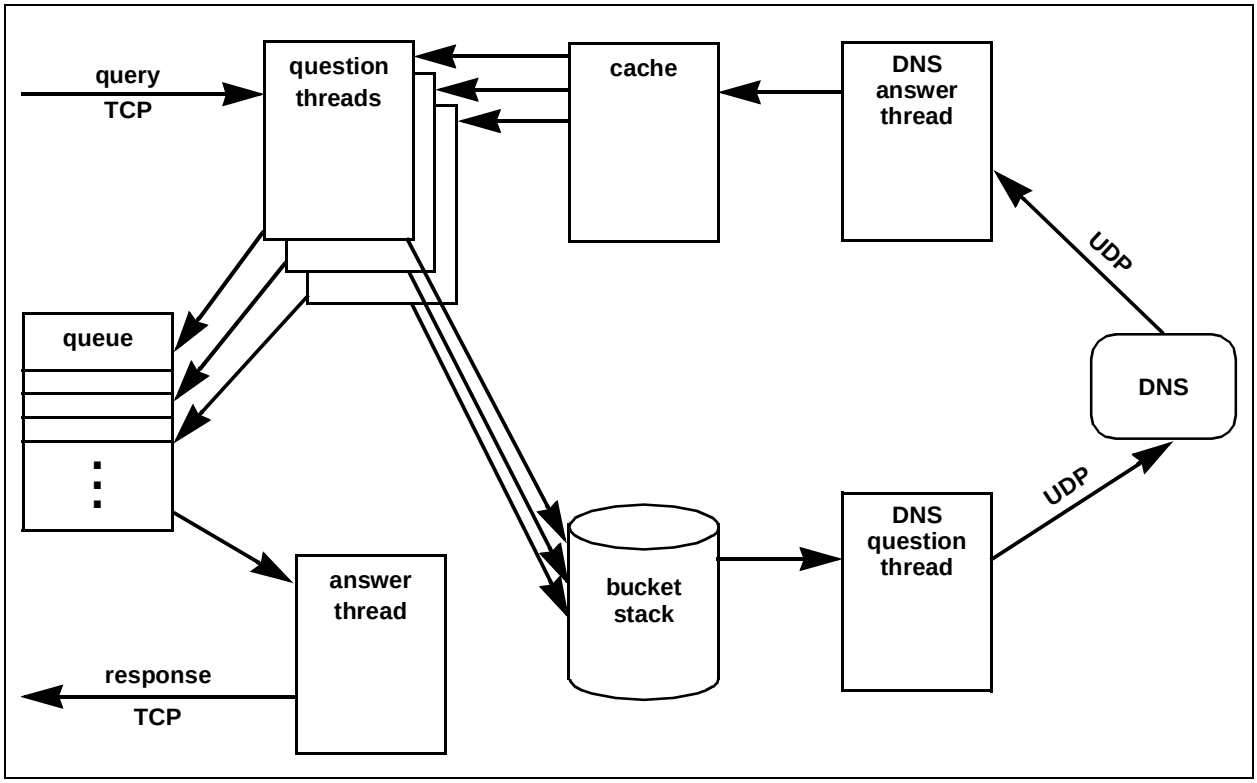


**Figure 2: Flow of Data through the Rapid DNS Server**

pling gives Rapid DNS the ability to provide quick results without waiting on answers from DNS servers.

The front and back ends are tied together with a cache and a stack. The cache, fed by the back end, contains all the DNS answers that the server has received. The front end reads from the cache to provide answers for client requests. If the cache does not contain the answer, then the front end answers “I don't know” and places the address on the stack. The back end drains the stack by sending questions to a DNS server. The stack is a fixed size, called a “leaky bucket”, and will be discussed in more detail in a later section.

The flow of data through the threads and data objects is depicted in Figure 2. A request is read by one of the question threads, which then performs a lookup for the requested IP address in the cache. Any entry found in the cache is used to answer the query: the IP address, its name, and a pointer identifying the i/o stream is placed on an answer queue. If no entry is found in the cache, then a null string is used for the answer and the IP address is placed in the DNS bucket. A fixed number of answer threads drains the answer queue by composing and sending out responses. The DNS question thread drains the DNS bucket, composing queries that ask for PTR records in the domain *in-addr.arpa*. These requests are sent to a name server via UDP. The DNS answer thread reads all DNS replies sent to the process, extracts the domain name and adds an entry to the cache.

A maintenance thread is run at periodic intervals to perform two functions: cache maintenance and persistent storage. Cache maintenance consists of a sweep through the cache to look for entries that have expired. At regular intervals the entire contents of the cache is written to a text file. This provides for a persistent record of the information and allows the cached data to survive server restarts. This file has other uses as well, since it contains address to host mappings of nearly all clients to visit the site in the past several days.

## 4.3 Protocol

The protocol used between client and server is extremely simple. Although its specification is not significant to the results presented here, a brief description is provided.

The protocol runs over a TCP stream, and an individual connection can handle an unlimited number of requests. A request (from client to server) consists of an IP address represented by 4 octets in network byte order (most significant byte first). Each IP address is separated from the next in the data stream with a framing

octet consisting of all 1's. Should the server get out of sync with the client, it will be able to resync within a few requests.

A response (from server to client) consists of an IP address followed by a null-terminated string. The IP address is formatted as in the request: 4 octets in network byte order. The string is the domain name associated with the IP address and ends with a zero octet. Each response is separated from the next with a framing octet consisting of all 1's. Responses are not coupled with requests: any number of requests can be sent between responses from the server.

There are two peculiarities in this protocol. First, the only identifying information in the response is the IP address itself. This is considered sufficient to match responses with requests, even though it is not unique per request. Second, there is no explicit length given for the variable length response. The client is expected to read until seeing the null octet, and the framing octet is used to ensure that client and server do not get out of sync.

## 5 Cache Management

The main cache holds answers received from DNS servers. As queries arrive from clients, the answers are served directly out of the cache. As more information is retained in the cache, the likelihood of a cache hit for a given request will increase. But the size of the cache cannot grow without bound due to system memory constraints. A variety of cache management techniques can be used to provide a trade-off between information retention and memory utilization. When we first began this project, we anticipated very high memory utilization on the part of the cache and sized our server system accordingly.

In the current implementation of Rapid DNS, the cache is implemented as a bucket hash keyed on IP address. The number of buckets is fixed throughout the lifetime of the server process, but can be configured at start-up time. The current configuration at CNN uses 400,009 buckets. Each bucket contains a linked list of items which is hashed to the bucket, and there is no limit on the length of each list. Since multiple threads of execution need to access the cache simultaneously, mutex locks are necessary to preserve the integrity of the data structure. Figure 2 clearly shows that only one thread adds data to the cache while multiple threads may be reading information from the cache. To optimize performance, read/write locks[8] were utilized within the cache with one lock being assigned to each bucket in the hash table. Any number of question threads can be reading information from the cache simultaneously. When the DNS answer thread needs to add an entry to the cache, it must first determine the target bucket in the hash table, then it must obtain a write lock on the bucket before inserting the new datum on to the bucket's linked list. A write lock on the bucket must also be obtained before any datum in the bucket is altered or removed.

Although the number of buckets for the cache is constant, the cache is not of a fixed size and a cache management algorithm must be employed to prevent unbounded growth. The current implementation utilizes a first-in first-out (FIFO) algorithm bounded by time. When an entry is added to the cache, it is stamped with an expiration time $x$ seconds in to the future. The configurable value $x$ is the "time to live" and is typically set between three and seven days. Lower settings for time to live result in a smaller run-time cache size. At periodic intervals, a maintenance thread is run that sweeps through the cache and removes any entry beyond its expiration time. The memory used by those entries is returned to the free pool so that it can be used for new entries. In the current implementation the configured time to live is the only value consulted when calculating the expiration time of an entry. The time to live value contained in the response from the DNS server is ignored. Although this policy may degrade the accuracy of the answer, it does reduce the server's dependency on data not under our direct control. If the server used DNS time to live values, then remote servers would have a direct influence on the cache hit ratio. In this application, we prefer an answer that is potentially a few days out of date over no answer at all. Although this policy may have an affect on the statistical results presented below, it certainly has no influence on the implementation. The server could easily be changed to take the minimum of the configured time to live and the DNS time to live when creating the cache entry.

The choice of a strict FIFO cache policy is primarily due to performance concerns. A policy that is tied to the entry creation time does not require modification of the entry at any point during its lifetime. Any policy based on use would require such usage to be tracked, and that question threads modify the entries they were reading to stamp them with a last use time. Such an implementation would require that these threads obtain a write lock before modifying the entry. With a FIFO policy, only the DNS answer thread and the maintenance thread need to obtain write locks. If all the question threads performed write locks, the lock contention would have a detrimental impact on performance.

## 6 DNS Request Bucket

When a cache miss occurs, the question thread inserts the missing IP address in to a stack. The back end DNS question thread pops addresses off this stack and composes DNS queries for the corresponding PTR records. This data structure is not implemented as a FIFO queue, but as a LIFO stack of bounded size. If a new request fills the stack, then requests at the bottom of the stack are dropped. This "leaky bucket" method prevents an ever increasing backlog of requests while avoiding swamped DNS servers.

Consecutive DNS requests are spaced with a configurable delay to avoid flooding the DNS server. The LIFO implementation intentionally gives priority to most recently received requests. It is expected that web page requests will have a high locality of reference but only for a brief period of time. This is due to two observations. First, the typical web page "view" consists of many embedded images, each of which will generate a separate HTTP request from the same client. Second, it is expected (hoped) that upon seeing the first page the user will be drawn in to the site and request additional pages. So we expect that the longer it has been since we've heard from a client the less likely it is that we will hear from it again in the near future.

The fixed size insures that the stack will not grow without bound. As newer requests enter the bucket, older ones are forced to wait. With each new request the likelihood that the request at the bottom of the bucket will get serviced decreases. Since requests at the bottom of the stack are least likely to ever get serviced they might as well be discarded. The impact of discarding a request in the bucket is that a future request may miss in the cache lookup rather than hit. The result is an "I don't know" answer, which is not a serious concern. As a consequence, however, the IP address will once again be entered at the top of the stack and will have another chance at getting serviced.

A request that misses in the cache will cause an entry to be placed in the bucket, but it might be some time before that request is filled by DNS. During that period of time many more requests for the same address may arrive. To help suppress duplicates in the bucket, the back end will create an empty entry in the cache for the address, giving it two minutes to live. When the back end receives the DNS answer, it simply replaces the null cache entry with the actual data and adjusts the TTL. If the back end sees a request in the bucket for an address that is already in the cache (even if the cache entry is empty), it will not send out a DNS query and instead log the event as a "back end cache hit". Collectively, this technique suppresses the creation of duplicate DNS queries.

## 7 Negative Caching

A good percentage of the responses from DNS indicate that the requested reverse domain does not exist (response code NXDOMAIN). Consider the consequences of this result on the Rapid DNS server if this result is ignored. After two minutes the blank cache entry which was inserted for duplicate suppression (see Section 6) will expire and additional requests for the address will generate another DNS query. Therefore we considered it prudent to cache this negative information. A response of "no such domain" is represented in the cache with a null entry, and its time to live is set to be the same as regular entries. Further requests for that entry will generate an "I don't know" answer. In this particular case, however, we actually do know that there is no name for this number. From the perspective of the client, the distinction between "not known" and "nonexistent" is unimportant, as they would be logged the same.

During production operation we have noted that a good percentage of the DNS requests generate a failure (response code SERVFAIL). In a typical week we measured that an average 12% of the DNS queries resulted in a SERVFAIL, with a peak at 19.4%. Currently we ignore such responses, and consequently the address that failed will be re-queried in as little as two minutes. The high percentage of such answers implies a potential performance benefit from caching them. Since this is considered to be a transient error, such entries would have to be cached with a much lower time to live than negative answers.

## 8 Performance Results

Each Rapid DNS server in the CNN web farm routinely handles 250 client connections. Many web servers in the CNN farm are configured to create multiple Netscape processes (or "instances"), and each process needs to open a separate connection. So although there are over 250 client connections on a Rapid DNS server, in reality it may only be serving 50 to 70 machines. The current installation utilizes three Rapid DNS servers, and the clients load balance across the servers. Each server is a Sun SPARC Ultra 2200 with two 200 MHz processors and 1 gigabyte of physical memory. Clients are configured to perform Rapid DNS queries only for http pages (image retrieval does not generate a query).

The cache carries approximately 2.5 million entries (negative information excluded) while the correspond-

ing executable consumes 260 megabytes of virtual memory. Due to the generous server configuration and Unix paging policies, all of the virtual memory typically remains resident. Round trip times for client requests average 2 milliseconds even during heavily loaded periods of the day. Measurements have shown request rates as high as 1,611,180 requests per hour. This corresponds to an average of 26,853 requests per minute or 447 per second.*

## 8.1 One Week with Standard Configuration

The standard configuration for Rapid DNS servers in the CNN Web Farm uses a hash table with 400,009 slots, a request bucket size of 4096, a 7-day time to live for cache entries, and a DNS query interval of 50 milliseconds. Negative caching is enabled in the standard configuration

Over the course of a one week measuring period with measurements taken hourly, the Rapid DNS server served an average of 5902 requests per minute (98 per second). The peak hourly rate was 862,500 requests, equivalent to 14,375 requests per minute (240 requests per second). During the same measuring period the cache hit ratio ranged from 86.5% to 94.5% with an average ratio of 92.4% and a standard deviation of 1.5%. The measuring period also saw no bucket leaks, except for those caused by a server restart in the middle of the week.

Hourly performance measurements are presented in Figure 3, where the top line shows requests and the lightly shaded area represents cache hits. The diurnal access behavior observed by Gribble and Brewer [5] is evident in these measurements: peak access times for the web servers are reflected in the quantity of Rapid DNS requests.

## 8.2 Varying the Query Interval

The rate at which the server sends out DNS queries, the query interval, can be tuned to avoid swamping the DNS server during peak loads. Initially this interval was set to 80 milliseconds, but it was discovered that this setting would cause bucket leaks during normal daytime loading.

The query interval defines the rate at which requests in the bucket are processed. If it is too slow the server will be forced to let requests leak out of the bucket. Smaller intervals may send too many requests to the DNS server with the result that some will never get answered. Given a query interval, one can easily compute an upper bound on the frequency of DNS queries that can be accommodated without the risk of leaks. This rate is simply the reciprocal of the frequency:

$$r(f) = \frac{1}{f}$$

Measurements show the expected behavior: a 90 millisecond delay results in a maximum rate of 11.11 requests per second. Given a cache hit ratio of 90% we would expect to see this rate when the number of client requests is approximately 111, a figure that is routinely exceeded during the day. As predicted, a configuration using 90 milliseconds saw a substantial number of bucket leaks. Figure 4 shows an average day running with a 90 ms interval. The top line depicts the number of requests, the lightly shaded area shows the cache hits, and the dark shading indicates the resulting number of bucket leaks. Figure 5 compares the number of actual DNS requests against the leaks for the same period of time. The ceiling on the request rate is evident, and the corresponding leaks show the expected behavior. Subsequent measurements using query intervals between 60 and 80 ms showed negligible leaks during an average week with an acceptable load on the DNS server.

## 8.3 Effects of Negative caching

To show the benefits of negative caching, we ran one server with negative caching disabled for a one-day period. Figure 6 shows a comparison between this server (the subject) and one which was still caching negative information (the control) for the same time period. Both servers were configured to use a query interval of 50 milliseconds.

It is immediately noticeable that the cache hit rate is significantly lower without negative caching. The subject machine had an average rate of 78.9% while the control saw 90.45%. The subject never exceeded a cache hit rate of 89% and saw a low of 61.2%, while the control had a much more compact range from 84.4% to 93.1%. A comparison of the cache hit rates is given in Figure 7. Because of the lower cache hit rate, more DNS queries enter the bucket and the configuration cannot get all of them out. Consequently there is a corresponding increase in the number of bucket leaks. The subject server peaked at 175,906 leaks in an hour where the control server experienced 75,309 in its worst hour.

Also of interest is the fact that the peak in number of requests for the subject is noticeably lower. This can be attributed to the load balancing code used in the client,

* During early development we ran with only two Rapid DNS servers, and it was common to see hourly rates between 1.5 and 1.6 million.

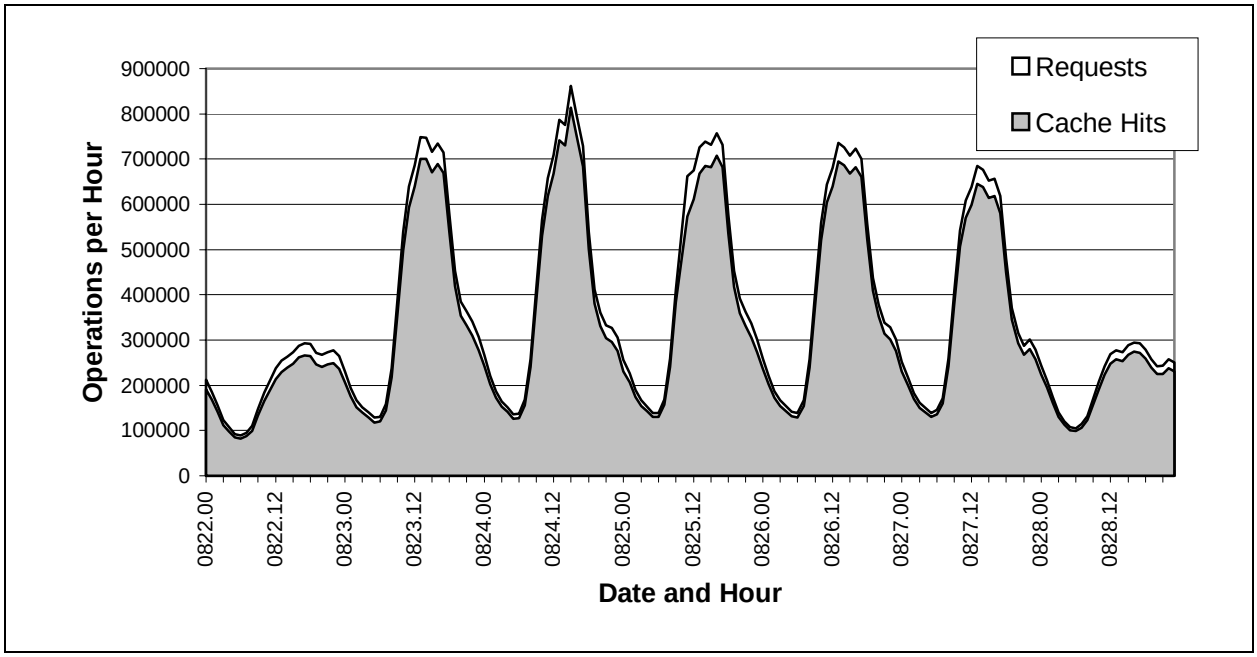


**Figure 3: Rapid DNS Server Performance for One Week**

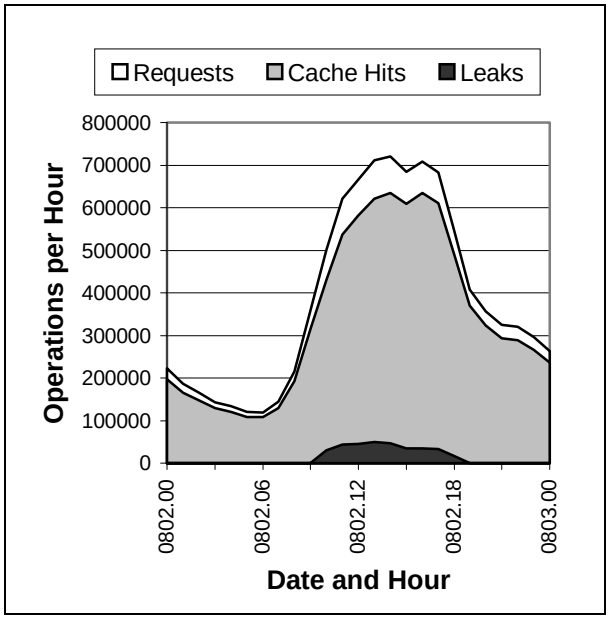


**Figure 4: Performance with 90 ms Query Interval**

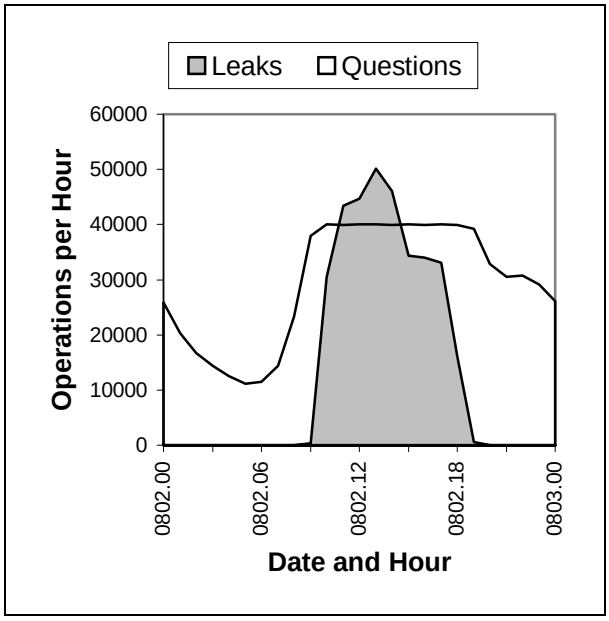


**Figure 5: DNS Questions with 90 ms Query Interval**

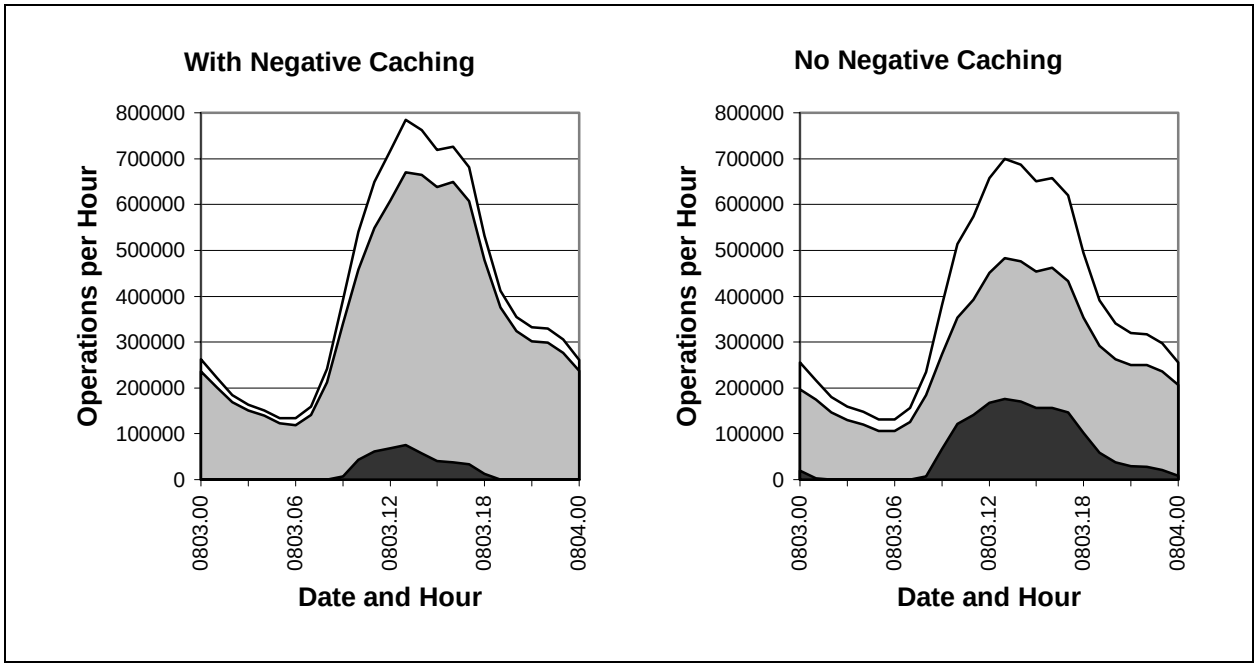


**Figure 6: Impact of Negative Caching on Server Performance**

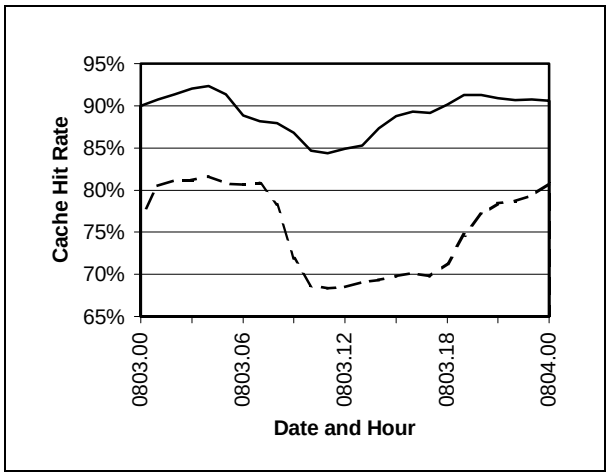


**Figure 7: Negative Caching Effects on Cache Hits**

which would imply that the average round trip time for the server without negative caching was higher. A higher round trip time would lead to a larger backlog of requests in the client, which in turn would schedule around the backlog and allocate fewer requests to the slower server. The implication is that the added workload caused by processing additional DNS queries and responses had an impact on its ability to provide timely answers to queries.

## 9 Conclusions

Rapid DNS has provided the CNN Web Farm with a viable mechanism for obtaining client domain names. This ability has enabled it to provide more accurate demographic information and to enable targeting of advertisements by domain name. It is unlikely that direct use of the DNS name servers would have been as effective, given the latency inherent in any domain lookup. The system has met and exceeded expectations since its deployment in March of 1999. One or two web server outages were attributed to the Rapid DNS service, but were caused by simple programming bugs and not by flaws in the design. The system has proven its viability over the course of six months.

Some observations have come out of analysis of the server data.

- The use of negative caching has a marked impact on performance of this particular application.
- Very high cache hit rates have been realized by the system, minimizing direct load on the DNS name servers.
- Even with over 250 client connections, the servers are able to sustain in excess of 400 operations per second.
- A built-in mechanism for throttling outgoing DNS queries controls the load on the name server with a negligible loss of data, especially using query intervals of 80 ms and lower.

## 10 Further Work

There are several areas that can benefit from further study. The justification for using a FIFO bucket to queue DNS queries should be tested by comparing the performance of different queueing policies. We suspect there would be a measurable improvement in the cache it ratio if SERVFAIL responses (Section 7) were cached, and a study to determine this would be beneficial. Ignoring the time to live field in the DNS reply (Section 5) sacrifices some accuracy to improve performance. The extent of the inaccurate information is not known, but could be easily measured. The very high cache hit ratios seen by this study imply a high locality of reference for web clients. It would be interesting to know if this is a phenomenon peculiar to CNN or if and to what extent this pattern is present at other sites.

## Availability

The code for this project was developed under contract with Cable News Network and remains proprietary. It is not available for public distribution.

## Acknowledgements

The authors would like to thank Steve Brunton for doing much of the dirty work in installing, baby sitting, and troubleshooting the servers. He performed every software and configuration change we requested without a word of complaint. Thanks are also extended to Paul Holbrook for inspiring us to submit the work. Many of the ideas that went in to Rapid DNS, especially the leaky bucket, are due to Monty Mullig and Sam Gassel.

## Bibliography

[1] D. Andresen, T. Yang, V. Holmedahl, O. H. Ibarra, "SWEB: Toward a Scalable World Wide Web Server on Multicomputers," *Proceedings of the 10th International Symposium on Parallel Processing*, pp. 850-856, April 1996.

[2] T. Brisco, "DNS Support for Load Balancing," RFC 1794, April 1995.

[3] V. Cardellini, M. Colajanni, P. Yu, "Redirection Algorithms for Load Sharing in Distributed Webserver Systems," Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.

[4] P. Danzig, K. Obracza, A. Kumar, "An Analysis of Wide-Area Name Server Traffic," *SIGCOMM '92 Conference Proceedings: Communications, Architectures and Protocols*, pp. 281-292, August 1992.

[5] S. Gribble, E. Brewer, "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[6] T. T. Kwan, R. McGrath, D. Reed, "NCSA's World Wide Web Server: Design and Performance," *Computer*, **28**(11), pp. 68-74, November 1995.

[7] P. Manning, P. Vixie, "Operational Criteria for Root Name Servers," RFC 2010, October 1996.

[8] P. McKenney, "Selecting Locking Primatives for Parallel Programming," *Communications of the ACM*, **39** (10), pp. 75-82, October 1996.

[9] P. Mockapetris, "Domain Names – Implementation and Specification," RFC 1035, November 1987.

[10] Netscape Communications Corporation, *Netscape Enterprise Server Programmers Guide for Unix*, 1996.

[11] R. Schemers, "lbnamed: A Load Balancing Name Server in Perl," *Proceedings of the Ninth Systems Administration Conference*, pp. 1-11, September 1995.