Serving DNS using a Peer-to-Peer Lookup Service

Russ Cox, Athicha Muthitacharoen, Robert T. Morris
MIT Laboratory for Computer Science
{rsc,athicha,rtm}@lcs.mit.edu

Abstract

The current domain name system (DNS) couples ownership of domains with the responsibility of serving data for them. The DNS security extensions (DNSSEC) allow verificaton of records obtained by alternate means, opening exploration of alternative storage systems for DNS records. We explore one such alternative using DHash, a peer-to-peer distributed hash table built on top of Chord. Our system inherits Chord's fault-tolerance and load balance properties, at the same time eliminating many administrative problems with the current DNS. Still, our system has significantly higher latencies and other disadvantages in comparison with conventional DNS. We use this comparison to draw conclusions about general issues that still need to be addressed in peer-to-peer systems and distributed hash tables in particular.

1 Introduction and Related Work

In the beginnings of the Internet, host names were kept in a centrally-administered text file, `hosts.txt`, maintained at the SRI Network Information Center. By the early 1980s the host database had become too large to disseminate in a cost-effective manner. In response, Mockapetris and others began the design and implementation of a distributed database that we now know as the Internet domain name system (DNS) [8, 9].

Looking back at DNS in 1988, Mockapetris and Dunlap [9] listed what they believed to be the surprises, successes, and shortcomings of the system. Of the six successes, three (variable depth hierarchy, organizational structure of names, and mail address cooperation) relate directly to the adoption of an administrative hierarchy for the names. The administrative hierarchy of DNS is reflected in the structure of DNS servers: in typical usage, an entity is responsible not only for maintaining name information about its hosts but also for the serving that information.

The fact that service structure mirrored administrative hierarchy provided a modicum of authentication for the returned data. Unfortunately, IP addresses can be forged and thus it is possible for malicious people to impersonate DNS servers. In response to concerns about this and other attacks, the DNS Security Extensions [5] (DNSSEC) were developed in the late 1990s. DNSSEC provides a stronger mechanism for clients to verify that the records they retrieve are authentic.

DNSSEC effectively separates the authentication of data from the service of that data. This observation enables the exploration of alternate service structures to achieve desirable properties not possible with conventional DNS. In this paper, we explore one alternate service structure based on Chord [10], a peer-to-peer lookup service.

Rethinking the service structure allows us to address some of the current shortcomings in the current DNS. The most obvious one is that it requires significant expertise to administer. In their book on running DNS servers using BIND, Albitz and Liu [1] note that many of the most common name server problems are configuration errors. Name servers are difficult and time-consuming to administer; ordinary people typically rely on ISPs to serve their name data. Our approach solves this problem by separating service from authority — clients can enter their data into the Internet-wide Chord storage ring and not worry about needing an ISP to be online and to have properly configured its name server.

DNS performance studies have confirmed this folklore. In 1992, Danzig *et al.* [4] found that most DNS traffic was caused by misconfiguration and faulty implementation of the name servers. They also found that one third of the DNS traffic that traversed the NSFNet was directed to one of the seven root name servers. In 2000, Jung *et al.* [6] found that approximately 35% of DNS queries never receive an answer or receive a negative answer, and attributed many of these failures to improperly configured name servers or incorrect name server (NS)

Russ Cox was supported by a Hertz Fellowship while carrying out this research.

records. The study also reported that as much as 18% of DNS traffic is destined for the root servers.

Serving DNS data over Chord eliminates the need to have every system administrator be an expert in running name servers. It provides better load balance, since the concept of root server is eliminated completely. Finally, it provides robustness against denial-of-service attacks since disabling even a sizable number of hosts in the Chord network will not prevent data from being served.

2 Design and Implementation

We have implemented a prototype of our system, which we call DDNS.

Our system handles lookups at the granularity of resource record sets (RRSets), as in conventional DNS. An RRSet is a list of all the records matching a given domain name and resource type. For example, at the time of writing, *www.nytimes.com* has three address (A) records: 208.48.26.245, 64.94.185.200, and 208.48.26.200. These three answers compose the RRSet for (*www.nytimes.com*, A).

DNSSEC uses public key cryptography to sign resource record sets. When we retrieve an RRSet from an arbitrary server, we need to verify the signature (included as a signature (SIG) record). To find the public key that should have signed the RRSet, we need to execute another DNS lookup, this time for a public key (KEY) RRSet. This RRSet is in turn signed with the public key for the enclosing domain. For example, the (*www.nytimes.com*, A) RRSet should be signed with a key listed in the (*www.nytimes.com*, KEY) RRSet. The latter RRSet should be signed with a key listed in the (*nytimes.com*, KEY) RRSet, and so on to the hierarchy root, which has a well-known public key.

DDNS stores and retrieves resource record sets using DHash [3], a Chord-based distributed hash table. DHash has two properties useful for this discussion: load balance and robustness.

DHash uses consistent hashing to allocate keys to nodes evenly. Further, as each block is retrieved, it is cached along the lookup path. If a particular record is looked up n times in succession starting at random locations in a Chord ring of m nodes, then with high probability each server transfers a given record only $\log m$ times total before every server has the record cached.

DHash is also robust: as servers come and go, DHash automatically moves data so that it is always stored on a fixed number of replicas (typically six). Because the replicas that store a block are chosen in a pseudo-random fashion, a very large number of servers must fail simultaneously before data loss occurs.

To create or update a DDNS RRSet, the owner prepares the RRSet, signs it, and inserts it into DHash. The key for the RRSet is the SHA1 hash of the domain name and the RRSet query type (*e.g.*, SHA1(*www.nytimes.com*, A)). DHash verifies the signature before accepting the data. When a client retrieves the RRSet, it also checks the signature before using the data.

Naively verifying a DNS RRSet for a name with n path elements requires n KEY lookups. We address this problem by allowing the owner to present additional relevant KEYs in the RRSet. To avoid inflating the responses, we can omit KEY RRSets for popular names. For example, the record containing the (*www.nytimes.com*, A) RRSet might also include the (*www.nytimes.com*, KEY) RRSet and the (*nytimes.com*, KEY) RRSet but omit the (*.com*, KEY) RRSet on the assumption that it would be widely cached. The key for the root of the hierarchy is assumed to be known by all DDNS servers, just as the IP addresses of the root servers are known in the current DNS.

To ease transition from conventional DNS to our system, a simple loopback server listening on 127.1 could accept conventional DNS queries, perform the appropriate Chord lookup, and then send a conventional response. Then systems could simply be configured to point at 127.1 as their name server.

3 Evaluation

To evaluate the use of Chord to serve DNS data, we used data from the study by Jung *et al.* on a simulated network with 1000 DDNS nodes. For the test, we turned off replication of records. We did not simulate node failures, so the only effect of replication would be to serve as *a priori* caching. Since the network is a fair amount smaller than the expected size of a real Chord ring, replication would bias the results in our favor because of the increased caching.

We first inserted answers to all the successful queries into the Chord network and then ran a day's worth of successful queries (approximately 260,000 queries), measuring storage balance, load balance while answering queries, and the number of RPCs required to perform each lookup. The distribution of names in the day's queries is heavy tailed.

To simulate failed DNS queries, we started a similar network, did not insert any data, and executed a day's worth of unresolved queries (approximately

220,000 queries), measuring load balance and RPC counts. These queries were distributed similarly to the successful queries.

Finally, to simulate popular entries, we ran what we called the "slashdot test," inserting one record and then fetching it a hundred thousand times.

All three tests began each lookup at a random node in the Chord network.

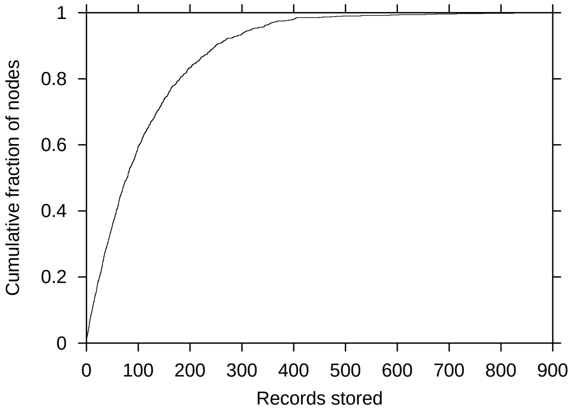

Figure 1: Storage balance for 120,000 records. The graph shows a cumulative distribution for the number of records stored on each node. Perfect balance would place the same number of records on each node, making the cumulative distribution a vertical line around 120.

For the successful queries test, we inserted approximately 120,000 records to serve as answers to the 260,000 queries. Figure 1 shows that the median number of records stored per node is about 120, as expected. The distribution is exponential in both directions because Chord nodes are randomly placed on a circle and store data in proportion to the distance to their next neighbor clockwise around the circle. Irregularities in the random placement cause some nodes to store more data than others. The two nodes that stored in excess of 800 records (824 and 827) were both responsible for approximately 0.8% of the circle, as compared with an expected responsibility of 0.1%. Even so, this irregularity drops off exponentially in both directions, and can be partially addressed by having servers run multiple nodes in proportion to their storage capacities. We conclude that DDNS does an adequate job of balancing storage among the peers.

Even though storage is balanced well, some records are orders of magnitude more popular than others. Since records are distributed randomly, we need to make sure that nodes that happen to be responsible for popular records are not required to serve a disproportionate amount of RPCs. DHash's block caching helped provide load balance as measured by RPCs served per lookup per node. As shown in Figure 2, in the successful query test, nodes served RPCs in approximate proportion to the number of records they stored. Specifically, each node serves each of its popular blocks about ten ($\log_2 1000$) times; after that, the block is cached at enough other nodes that queries will find a cached copy instead of reaching the responsible server. A similar argument shows that very quickly every node has a copy of incredibly popular blocks, as evidenced by the Slashdot test: after the first few thousand requests, virtually every node in the system has the record cached, so that subsequent requests never leave the originating node.

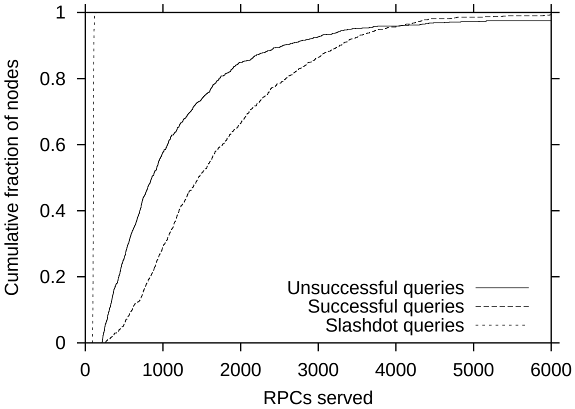

Figure 2: Load balance. The graph shows a cumulative distribution for the number of RPCs served per lookup by each node during the test. Since there are a thousand nodes, ideal behavior would be involving each node in $n/1000$ RPCs, where n is the median number of RPCs executed per lookup (see Figure 3).

For the unsuccessful query test, nodes served RPCs in proportion to the number of queries that expected the desired record to reside on that node. This does a worse job of load balancing since there is no negative caching.

The graphs shows that the loads are similar for both successful and unsuccessful queries, unlike in the current DNS, where unresponsive queries might result in spurious retransmissions of requests.

Figure 3 shows the number of RPCs required by a client for various lookups. Successful queries and unsuccessful queries have the same approximately random distribution of hop counts, except that successful queries usually end earlier due to finding a cached copy of the block. Since the slashdot record got cached everywhere very quickly, virtually all lookups never left the requesting node.

We would like to be able to compare the latency for DNS over Chord with the latency for conventional DNS.

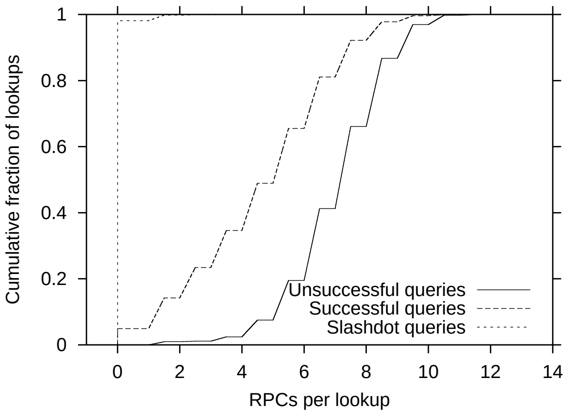

Figure 3: Client load to perform lookups.

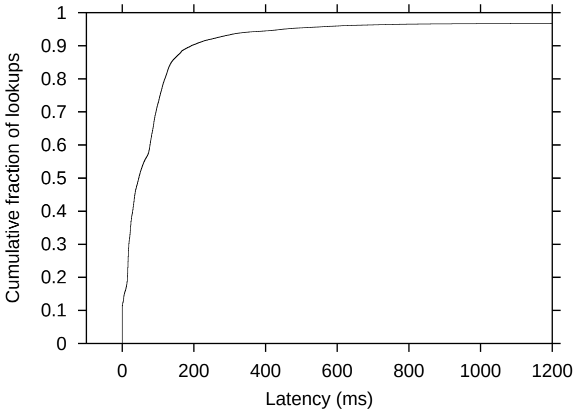

Figure 4: Lookup latency distribution for successful one-server queries over conventional DNS in the Jung. *et al.* data. Of the 260,000 successful lookups, approximately 220,000 only queried one server.

This is made difficult by the fact that we do not have an Internet-wide Chord ring serving DNS records. To compensate, we took the latency distribution measured in the Jung. *et al.* study and used it to compute the expected latencies of DNS over Chord in a similar environment.

Figure 4 shows the distribution of latency for successful lookups in the Jung. *et al.* one-day DNS trace. Because we don't have individual latencies for requests that contacted multiple servers, only requests completed in one round trip are plotted. The tail of the graph goes out all the way to sixty seconds; such slow servers would not be used in the Chord network, since timeouts would identify them as having gone off the network. Since such a small fraction of nodes have such long timeouts, we cut the largest 2.5% of latencies from the traces in order to use them for our calculations. To be fair, we also cut the smallest 2.5% of the latencies from the traces, leaving the middle 95%.

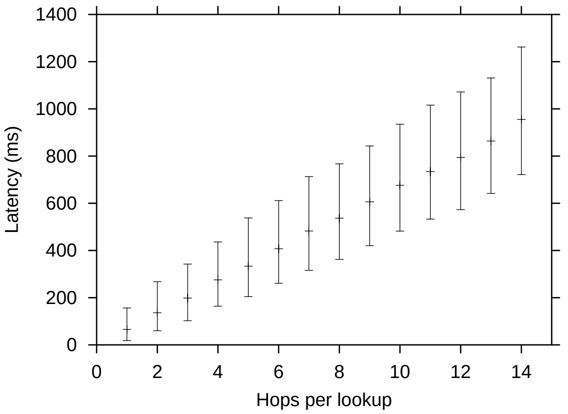

Figure 5: Hops per lookup in the simulation of the Jung. *et al.* traces over Chord.

Figure 5 shows the correlation between hops per lookup and expected latency. For each hop count h, we randomly selected and summed h latencies from the Jung. *et al.* distribution. Each point in the graph is the average of 1000 random sums; the bars show the standard deviations in each direction.

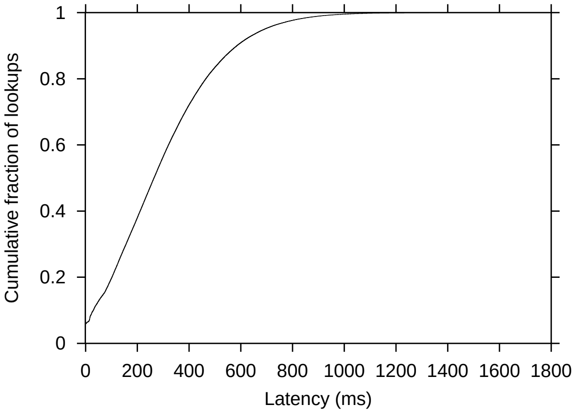

Figure 6: Imaginary latency to perform DNS lookups over Chord, assuming the latency distribution from the Jung. *et al.* trace plotted in Figure 4

Figure 6 displays the expected latency in another way. Here, for each query with hop count h, we chose a random latency (the sum of h random latencies from the Jung. *et al.* distribution) and used that as the query latency. The cumulative distribution of these query latencies is plotted.

These experiments show that lookups in DDNS take longer than lookups in conventional DNS: our median response time is 350ms while conventional DNS's is about 43ms. We have increased the median in favor of removing the large tail: lookups taking 60s simply cannot happen in our system.

4 Why (not) Cooperative DNS?

Serving DNS data using peer-to-peer systems frees the domain owners from having to configure or administer name servers. Anyone who wants to publish a domain only needs the higher-level domain to sign her public key. DDNS takes care of storing, serving, replicating, and caching of her DNS records. Below, we discuss various issues in the current DNS, and the extent to which DDNS solves them.

4.1 DNS Administration

In their 1988 retrospective [9], Mockapetris and Dunlap listed “distribution of control vs. distribution of expertise or responsibility” as one of the shortcomings of their system, lamenting:

> Distributing authority for a database does not distribute corresponding amounts of expertise. Maintainers fix things until they work, rather than until they work well, and want to use, not understand, the systems they are provided. Systems designers should anticipate this, and try to compensate by technical means.

To justify their point, they cited three failures: they did not require proof that administrators had set up truly redundant name servers without a single point of failure; in the documentation, they used hour-long TTLs in the examples but suggested day-long TTLs in the text, with the result that everyone used hour-long TTLs; debugging was made difficult by not requiring servers to identify their server version and type in an automated way. DDNS eliminates much of the need for expertise by automatically providing a routing infrastructure for finding name information, automatically avoiding single points of failure.

In their handbook for the Berkeley DNS server, BIND, Albitz and Liu [1] listed what they believed to be thirteen of the most common problems in configuring a name server. Our system addresses six of them.

Slave server cannot load zone data. DDNS solves this by automatically handling replication via the DHash protocol. There are no slave servers.

Loss of network connectivity. DDNS is robust against server failure or disconnection. Unfortunately, it suffers from network partitions. For example, if a backhoe cuts MIT from the rest of the Internet, even though hosts on the Internet will not see a disruption in any part of the name space (not even MIT’s names), hosts at MIT may not even be able to look up their own names! This problem is not new, since a client machine with no knowledge of MIT will require access to the root name servers to get started. Both systems partially avoid this problem with caching: popular DNS data about local machines is likely to be cached and thus available even after the partition. DDNS actually works better in this situation, since the remaining nodes will form a smaller Chord network and pool their caches. If proximity routing is deployed in Chord, then DDNS can use that make some probabilistic guarantee that each record for a domain name gets stored on at least one node close to the record owner.

Missing subdomain delegation. In conventional DNS, a domain is not usable until its parent has created the appropriate NS and glue records and propagated them. DDNS partially eliminates this problem, since there are no NS records. In their place, the domain’s parent would have to sign the domain’s public key RRSet. At the least, this eliminates the propagation delay: once the parent signs a domain’s public key, it is up to the domain’s administrator to publish it.

Incorrect subdomain delegation. In conventional DNS, if the parent is not notified when name servers or IP addresses of a domain change, clients will eventually not be able to find the domain’s name servers. The analogue in DDNS would be a domain changing its public key but forgetting to get its parent to sign the new key. Without getting the signature, though, inserts of records signed with the new key would fail. This would alert the administrator to the problem immediately. (In conventional DNS, the problem can go undetected since the local name server does not check to see whether the parent domain correctly points at it.)

On a similar note, Jung *et al.* [6] reported 23% of DNS lookups failed to elicit any response, partially due to loops in name server resolution. 13% of lookups result in a negative response, many of which are caused by NS records that point to non-existent or inappropriate hosts.

Conventional DNS requires that domain owners manage two types of information: data about hosts (*e.g.*, A records) and data about name service routing (*e.g.*, NS records). The latter requires close coordination among servers in order to maintain consistency; in practice this coordination often does not happen, resulting in broken name service. DDNS completely eliminates the need to maintain name service routing data: routing information is automatically maintained and updated by Chord without any human intervention.

In summary, we believe that using a peer-to-peer system for storing DNS records eliminates many common administrative problems, providing a much simpler way to serve DNS information.

4.2 Dynamically generated records

Our system requires that all queries can be anticipated in advance and their answers stored. Since the `hosts.txt` approach required this property and the original DNS papers are silent on the topic, it seems likely that this requirement was never explicitly intended to be relaxed. However, the conventional DNS did relax the requirement: since domains serve their own data, all possible queries need not be anticipated in advance as long as there is some algorithm implemented in the server for responding to queries. For example, to avoid the need to publish internal host names, the name server for *cs.bell-labs.com* will return a valid mail exchanger (MX) record for any host name ending in *.cs.bell-labs.com*, even those that do not exist.

Additionally, responses can be tailored according to factors other than the actual query. For example, it is standard practice to randomly order the results of a query to provide approximate load balancing [2]. As another example, content distribution networks like Akamai use custom DNS responses both for real-time load balancing and to route clients to nearby servers [7].

The system we have described can provide none of these capabilities, which depend on the coupling of the administrative hierarchy and the service structure. If some features were determined to be particularly desirable, they could be implemented by the clients instead of the servers.

4.3 Denial of Service

DDNS has better fault-tolerance due to denial-of-service attacks over the current DNS. Because there is no name server hierarchy, the attacker has to take down a diverse set of servers before data loss becomes apparent.

Another type of denial of service is caused by a domain name owner inserting a large number of DNS records, using up space in the Chord network. We can address this problem by enforcing a quota on how much data each organization can insert depending on how much storage the organization is contributing to DDNS.

5 Conclusions

Separating DNS record verification from the lookup algorithm allows the exploration of alternate lookup algorithms. We presented DDNS, which uses a peer-to-peer distributed hash table to serve DNS records. In our judgement, using DDNS would prove a worse solution for serving DNS data than the current DNS. We believe that the lessons we draw from comparing the two have wider applicability to peer-to-peer systems in general and distributed hash tables in particular.

DDNS eliminates painful name server administration and inherits good load balancing and fault tolerance from the peer-to-peer layer. The self-organizing and adaptive nature of peer-to-peer systems is a definite advantage here, something that conventional manually administered systems cannot easily provide.

DDNS has much higher latencies than conventional DNS. The main problem is that peer-to-peer systems typically require $O(\log_b n)$ RPCs per lookup. Chord uses $b = 2$, requiring 20 RPCs for a million node network. Systems such as Pastry and Kademlia use $b = 16$, requiring only 5 RPCs for a million node network. Our experiments show that using even 5 RPCs results in a significant increase in latency, and of course the problem becomes worse as the peer-to-peer network grows. By contrast, conventional DNS typically needs 2 RPCs; it achieves its very low latency by putting an enormous branching factor at the top of the search tree — the root name servers know about millions of domains. It is easy to hide this problem in the big-O notation, but if peer-to-peer systems are to support low-latency applications, we need to find ways to reduce the number of RPCs per lookup.

DDNS has all the functionality of a distributed `hosts.txt`, but nothing more. Conventional DNS augments this functionality with a number of important features implemented using server-side computation. Distributed hash tables aren't sufficient for serving DNS because they require features to be client-implemented. It is cumbersome to update all clients every time a new feature is desired. At the same time, in a peer-to-peer setting, it is cumbersome to update all servers every time a new feature is desired. This is one case where the enormous size of peer-to-peer networks is not offset by the self-organizing behavior of the network. Perhaps we should be considering “active” peer-to-peer networks, so that new server functionality can be distributed as necessary.

Finally, DDNS requires people publishing names to rely on other people's servers to serve those names. This is a problem for many peer-to-peer systems: there is no incentive to run a peer-to-peer server rather than just use the servers run by others. We need to find models in which people have incentives to run servers rather than just take free rides on others' servers.

References

[1] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly & Associates, 1998.

[2] T. Brisco. DNS support for load balancing. RFC 1794, April 1995.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[4] P. Danzig, K. Obraczka, and A. Kumar. An analysis of wide-area name server traffic: A study of the internet domain name system. In *Proc ACM SIGCOMM*, pages 281–292, Baltimore, MD, August 1992.

[5] D. Eastlake. Domain name system security extensions. RFC 2535, March 1999.

[6] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop '01*, San Francisco, California, November 2001.

[7] R. Mahajan. How Akamai works. http://www.cs.washington.edu/homes/ratul/akamai.html.

[8] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, November 1987.

[9] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proc. ACM SIGCOMM*, Stanford, CA, 1988.

[10] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, San Diego, 2001.