UNDERSTANDING INTERNET NAMING: FROM THE MODERN DNS ECOSYSTEM TO NEW DIRECTIONS IN NAMING

by

TOM CALLAHAN

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May 2013

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

TOM CALLAHAN

candidate for the **DOCTOR OF PHILOSOPHY** degree*

Committee Chair: Michael Rabinovich

Committee Member: Mark Allman

Committee Member: Vincenzo Liberatore

Committee Member: Soumya Ray

Committee Member: Frank Merat

Date: 03/25/2013

*We also certify that written approval has been obtained for any propriety material contained therein.

Contents

List of Tables

List of Figures

Acknowledgments

Many thanks to those who have supported me throughout this process, and without whom this thesis would not be possible:

- Mark Allman and Misha Rabinovich, for their tireless help and patience
- The International Computer Science Institute and the Case Connection Zone for providing datasets used within this dissertation
- My thesis committee
- My wife & family, for their support

TOM CALLAHAN

Case Western Reserve University

May 2013

Understanding Internet Naming: From the Modern DNS Ecosystem to New Directions in Naming

Abstract

by

TOM CALLAHAN

In this dissertation we study current and proposed implementations of Internet naming schemes. Creating reliable naming systems requires an up-to-date understanding of the current system's real-world operation. Therefore, our first goal is to understand the modern client-side Domain Name System (DNS) ecosystem through empirical study of both system components and operational DNS traffic. Next, we discuss two naming scenarios where the DNS is insufficient and propose solutions. One problem we examine—the second component of this dissertation—is that Internet transactions need a well-known rendezvous point to establish communication, often a DNS name. However, the static nature of these rendezvous points introduces brittleness to the process. Controversial rendezvous points (such as IP addresses and hostnames used to bootstrap into peer-to-peer networks) are often targeted by censors. Many factors can prevent the usage of a central hub in the absence of any adversary—central hubs can also be vulnerable to network failures, power failures, and human error. Rendezvous schemes based upon DNS are additionally vulnerable to failures of authoritative DNS servers and lapses in domain registration. Therefore, we design and evaluate a system that allows users to communicate without any centralized hub or fixed rendezvous point. Another problem we examine—and the third thrust of this dissertation—is that DNS does not encourage user-to-user information sharing in general. Publishing DNS records remains limited to systems administrators through often manual processes. Furthermore, we note that in general the DNS is used to name hosts, which most users are not interested in. Rather, users typically share content and pointers to other people on different Internet services; however, no DNS types exist to publish the Uniform Resource Locator (URL) for a user's webpage or an Instant-Messaging screen name used by a user. Therefore, we discuss a new naming system centered around users, allowing for secure publication and consumption of names by users and their applications.

Chapter 1

Introduction

Naming is an important component of creating usable computer networks. Without meaningful human-readable names such as "google.com" or "amazon.com", the World Wide Web [BLCL$^+$94] could not be as popular as it is today. The first Internet-wide naming schemes involved the dissemination of centrally-maintained lists of Internet hosts [Kud74, HSF85] and provided only simple mappings for host names to Internet addresses. Later, the Domain Name System (DNS) [Moc87b, Moc87c] both expanded the types of mappings that could be stored and allowed for late-binding – that is, allowing the mapping for a name to be provided dynamically at query time. Furthermore, the hierarchical design of DNS allows name owners to directly control their own namespace, as well as re-delegate portions thereof to other parties (thus creating new zones). As part of this design, each "zone" of authority has its own set of name servers, enhancing scalability. We note that the DNS continues to be the primary system for mapping Internet names to Internet addresses today. In this dissertation, we study three distinct areas: understanding the current DNS ecosystem, enabling decentralized communication, and a clean-slate naming system allowing users to easily name content and hold other meta-information.

Understanding the Modern DNS Ecosystem: DNS, like the Internet, is a constantly evolving system. While the DNS's initial purpose was simply to provide hostname lookups, the system has evolved to provide load balancing, geographically-sensitive traffic distribution, and non-hostname lookup services (such as DNS blacklists). Further, the hierarchy of the DNS allows the administrator of any zone to use the DNS for any conceivable service that complies with the DNS protocol. The changing nature of DNS means that researchers must keep our understanding of the DNS up-to-date. In order to characterize modern DNS traffic, we utilize a two-pronged empirical strategy, as follows.

Our first goal is to understand how client-side DNS resolution works. In particular, we aim to answer two questions—what devices are involved in the DNS resolution process? and how might these devices color the resolution process?. As DNS resolver topology and behavior may vary by the ISP resolvers and client devices used, we must study a variety of different resolution paths to understand the breadth of behaviors on the Internet. In Chapter 4, we probe over 1 million open DNS resolvers, most of which we determine to be consumer devices, in order to study the security, protocol compliance, caching behavior, and ISP meddling in the client-side DNS infrastructure. Active probing allows for many DNS devices on the Internet to be explored quickly, thereby lending breadth to our study. Furthermore, active probing allows us to uniformly probe the address space to explore DNS resolver characteristics of our own choosing. By constructing queries that will eventually arrive at our own authoritative DNS (ADNS) server, we are able to measure behavior of both the device being probed and the device querying our ADNS. Furthermore, by providing DNS responses ourselves, we can detect tampered results.

Our second goal is to understand the nature of DNS traffic on the Internet. In particular, we wish to understand the kinds of requests made by clients, the nature of the answers returned, and the ways in which clients use those answers. Real DNS traffic patterns can inform design decisions not only in DNS devices, but also in middleware devices and applications dealing with protocols relying on DNS. While passive monitoring is limited in the breadth of vantage points available, it (unlike active measurement) allows us to examine traffic generated by real users. In Chapter 5, we examine traffic generated by users of the “Case Connection Zone” (CCZ), an experimental network that connects roughly 90 homes in a neighborhood adjacent to Case Western Reserve University to the Internet via bi-directional 1 Gbps fiber links.

In sum, examining both facets of many resolvers around the world along with users’ real world DNS experience contributes to a basic understanding of the modern DNS. Our work provides others with an understanding of the behaviors of many client-side DNS resolution chains as well as a glimpse into the behavior of DNS clients themselves.

Enabling Decentralized Communication: Rendezvous is a fundamental component of computer networking. In particular, for two actors to engage in communication one actor must be able to either directly contact the other, or leave a message for the other to find. One simple method of rendezvous is DNS—application designers can create a DNS name that provides one or more addresses for others to connect to in order to initiate communication. However, this straightforward method suffers from its centralized nature. In particular, any actor with control over a user’s DNS path or with control of a DNS zone superior to the DNS name being used may effectively censor or disrupt the rendezvous process. Therefore, we aim to develop a method of rendezvous that does not rely on any central hub or DNS name that could be disabled by a censor. Furthermore, we note that the use of a central hub introduces brittleness into the system—network failures, lapses in domain registration, and other factors can prevent usage of the central hub even in the absence of any censors and/or policies.

One important instance of the rendezvous problem is bootstrapping peer-to-peer (P2P) networks. P2P bootstrapping is the process that allows a new P2P node to join a P2P network by connecting to another node already on the network. While the distributed and redundant nature of many P2P networks is robust to network failures, simple DNS rendezvous is not robust in the face of network failures affecting the DNS server. Therefore, nodes not already in the P2P network may have difficulty joining.

In Chapter 6, we contribute a mechanism that allows for rendezvous between pre-arranged actors without reliance on centralized infrastructure. We utilize the millions of open DNS resolvers on the Internet not as components of the hierarchical DNS system, but as independent storage devices. Our system allows a client to publish a small amount of information using a shared secret on one of these open DNS servers, and then enables any other client possessing the secret to both find the server upon which the message is stored and decode the message itself. To store data, we manipulate the cache of a DNS server using randomly generated domain names. Furthermore, we determine the DNS servers used by probing at runtime. Therefore, our system does not rely upon any specific DNS server, DNS name, or other predefined rendezvous point.

New Directions in Naming: While we have shown that the DNS may be extended to accommodate additional use cases, DNS is not suitable for many use additional applications. For example, while average users frequently consume information from the DNS, they rarely contribute mappings into the system. This has led to names controlled by service providers instead of by users themselves. For instance, names such as “trc36@case.edu” and “www.eecs.case.edu/$\sim$trc36” are ultimately controlled by “case.edu”. Also, once these names have been distributed to a user’s contacts (typically in an ad-hoc manner), changing these names requires action by each of those contacts. This “lock-in” is both onerous in the face of change and can distort the user incentives in choosing service providers. DNS itself is insufficient to solve this lock-in problem as underlying services rely on those provider-specific names.

We note that today’s Internet is characterized by information-sharing between users. Instead of merely using the Internet for email and looking up information, users today interact via photo/video distribution, blogs, audio/video chats, instant messaging, and more. Not only are the identifiers used in these applications provider-specific, they are also totally independent. Two users communicating via instant message will almost certainly have to manually exchange identifiers to communicate via a video chat or social media. We also note that not only are users sharing an increasing amount of information, they are doing so from an increasing variety of devices. A single user might access the Internet through a desktop PC, a laptop, a smartphone, and a tablet. In general however, a user must configure each device manually, despite using the same services on each device. We see an opportunity for users and/or applications to name commonly used meta-information for simple re-use. For instance, a user configuring an email account on a desktop PC should be able to use that account on a mobile phone with ease.

One could envision using DNS to alleviate this mess by creating a domain for each user and propagating these identifiers by standard names and types. However, this approach is insufficient for several reasons. First, the DNS lacks any privacy capabilities. Second, the amount of meta-information a user might need to store about a particular service may exceed the size limits commonly in place within the DNS infrastructure. Third, DNS has only limited capabilities for clients to dynamically update records.

In Chapter 7, we propose a clean-slate system to provide a layer of abstraction for any type of object that is centered around users. Each user in our system possesses a *collection* of *records*. Collections are analogous to DNS zones, however the lack of hierarchy in our system makes users the ultimate authorities for their collections. Users and their applications may securely publish named and typed records with arbitrary meta-information that may then be consumed by others in a user-centric manner. For instance, Alice might create a record named “work” with a type of “email” and a value of “alice@company.com” and make this record available to Bob, who would then be able to type “Alice:work” into his email client to send email to Alice’s work address. This scheme avoids lock-in, as Alice may change the target of her mapping at any time. Our system is built with security and flexibility as core design principles.

Chapter 2

DNS Overview

In this chapter, we summarize the operation of the Domain Name System (DNS) [Moc83a, Moc83b, Moc87b, Moc87c]. The primary purpose of the DNS is to map human-friendly Internet hostnames such as **www.case.edu** to Internet addresses, such as **129.22.104.136**. In addition, the DNS serves records that allow clients to locate network services, such as Internet mail servers or Microsoft Windows domain controllers. The DNS is also hierarchical – the namespace is divided such that an organization directly controls its own portion of the namespace. For instance, ICANN controls the root of the DNS namespace and delegates all names ending in ".edu" to Verisign[1], who in turn delegates all names ending in ".case.edu" to Case Western Reserve University, who finally delegates all names ending in ".eecs.case.edu" to its department of Electrical Engineering and Computer Science. This chapter is not meant to be a comprehensive discussion of the DNS, but rather a general overview sufficient for understanding the remainder of this dissertation.

2.1 DNS Components

The DNS is divided into three main components:

[1]Administrative control of the .edu TLD is actually held by eduCause [edu], however by agreement Verisign is responsible for the technical operations of the TLD.

- **DNS Namespace**: The global and hierarchical organization of zones and the records contained therein. Each zone is a portion of the DNS Namespace delegated to some administrator(s).
- **Authoritative DNS servers (ADNS)**: DNS servers that serve records for some zone.
- **DNS Resolvers**: Actors that query zones' ADNS as needed to resolve names in the namespace.

In Figure 2.1, we show a graphical representation of a portion of the DNS namespace. At the core of the DNS namespace is the root zone, which has authority for all names in the system. Responsibility for the root zone is given to a fixed set of 13 well-known Internet addresses, called root DNS servers (these are also ADNS). However, the root DNS servers do not maintain immediate control over most records; instead the namespace is divided into several zones, each with a responsible ADNS. In Figure 2.1, we show how the root servers *delegate* control over all names ending in **.edu** to servers operated by Verisign[2], and how those servers further *delegate* control over all names ending in **.case.edu** to to servers operated by CWRU. We represent each *delegation* in Figure 2.1 with an arrow; we note that each *delegation* creates a new zone, represented by by the ovals. Domains falling immediately below the root are referred to as "Top-Level" domains (TLDs), while domains falling immediately below TLDs are called "Second-Level" domains (or SLDs).

2.2 DNS Resolution

In Figure 2.2, we show the DNS resolution process on behalf of a user. At each phase of the process, we show an arrow indicating the direction in which a message is being sent, with a number representing the sequence in which all messages were sent. The process is as follows:

[2]As of the writing of this dissertation; this arrangement could be changed in the future.

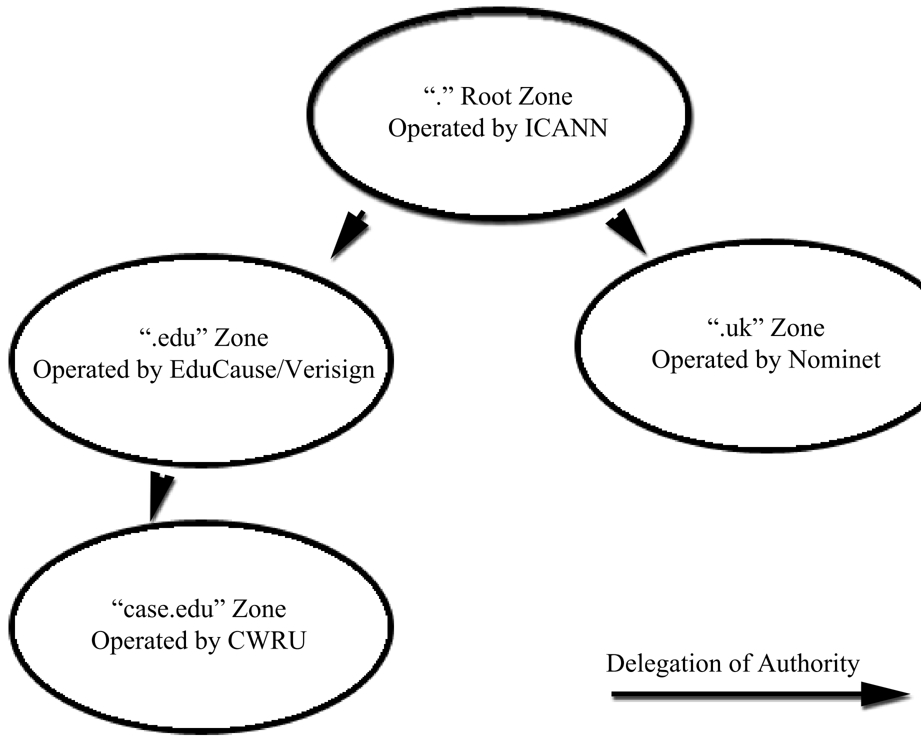


Figure 2.1: DNS Namespace

1. The client queries the local recursive resolver (RDNS) for the IP address of www.case.edu.

2. The RDNS visits a root ADNS and queries for the address of www.case.edu.

3. The root ADNS server indicates to the RDNS that it has delegated control for the .edu domain, and gives the RDNS the addresses of servers for .edu.

4. The RDNS visits the ADNS for the .edu domain (provided by the root server) and queries for the address of www.case.edu.

5. The ADNS for the .edu domain indicates to the RDNS that it has delegated control for the case.edu domain, and gives the RDNS addresses of servers for case.edu.

6. RDNS visits the ADNS for the case.edu domain and queries for the address of www.case.edu.

7. The case.edu ADNS is authoritative for the www.case.edu name, and therefore returns the IP address for www.case.edu to the RDNS.

8. the RDNS returns the address for www.case.edu to the client.

If at any step of the process the name being queried does not exist, the ADNS will return a response that does not contain an answer entry, but has a flag known as "NXDOMAIN" set. This is also known as a "negative answer". Such responses may be cached; this practice is referred to as "negative caching" or "NXDOMAIN caching".

We note that the records exchanged at every step in this process have the potential to be cached. Every DNS record has a field known as the time-to-live, or TTL, which specifies the maximum number of seconds that a record may be used before again querying the ADNS responsible for the record. However, the administrator of any device caching DNS records may make configuration choices affecting the duration a record stays in the cache. In the above example, if the RDNS had recently already asked for **abc.case.edu**, the RDNS would be able to elide steps 2-5. We also note that the client uses the same protocol to interact with the local RDNS server as the RDNS server uses with ADNS; many DNS software distributions include both RDNS and ADNS functionality. Due to the load incurred, as well as potential security concerns, it is usually not desirable to enable RDNS functionality on a public ADNS server.

2.3 DNS Records and Names

In Table 2.1 we show the format of DNS Resource Records (RRs). The first field of any record is the name field, which contains the fully-qualified domain name (FQDN) of the record. We call a name "fully-qualified" when all components of the name are included. For example, the name "www" may refer to "www.case.edu" when using a computer at Case Western Reserve University, however "www" will undoubtedly refer to another name

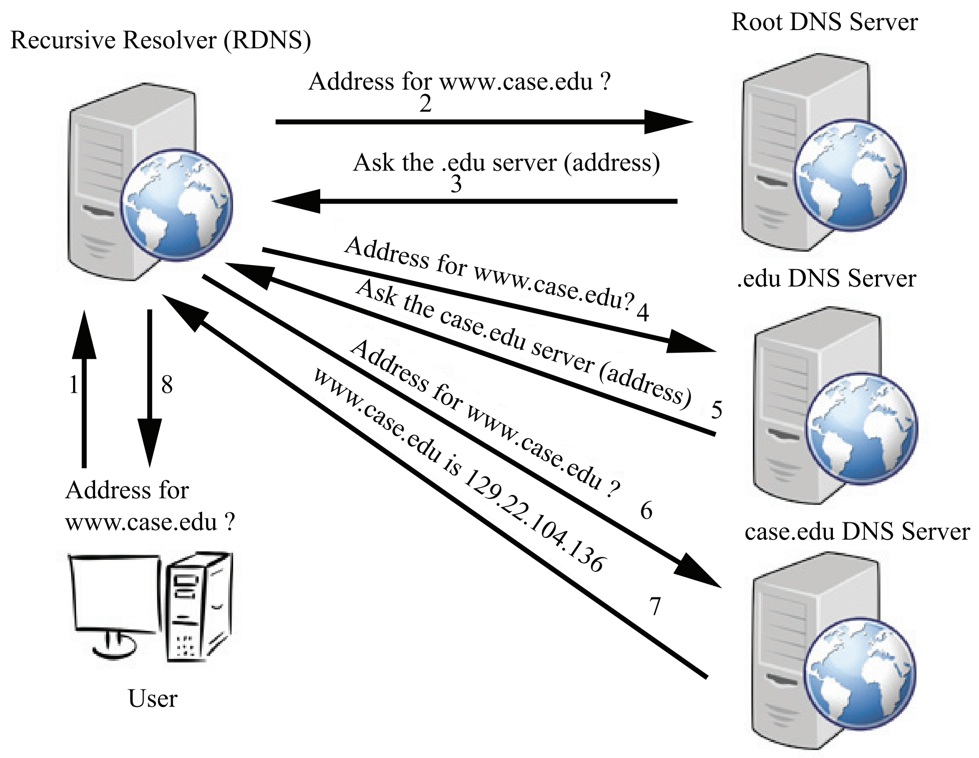

Recursive Resolver (RDNS)
Root DNS Server
Address for www.case.edu ?
2
Ask the .edu server (address)
3
Address for www.case.edu?
4
.edu DNS Server
Ask the case.edu server (address)
5
Address for www.case.edu ?
6
www.case.edu is 129.22.104.136
7
case.edu DNS Server
1
8
Address for
www.case.edu ?
User

Figure 2.2: DNS Resolution Process

at another location. In this example, "www.case.edu" is the FQDN, as it contains each name up to the root of the DNS hierarchy and is therefore unambiguous within the system.

The type field of a DNS RR is one of several well-known types; some more common types include "A" (IPv4 address), "MX" (mail exchanger), "NS" (nameserver), and "AAAA" (IPV6 address). The class field of a DNS RR is almost always the "IN", or Internet type. The TTL field allows a zone administrator to specify for how many seconds a record may be considered valid. For example, a TTL of 86,400 seconds on the **eecs.case.edu** domain specifies that the address given may be used for up to one day. We note that when a DNS RR is served from a well-behaved cache (such as that in an organization's RDNS), the TTL value given to a client is equal to the original TTL minus the number of seconds the record has been in the cache. In this way, the intended expiration of the record is preserved. Finally, the RDLength and RData fields describe the data actually transmitted, based upon the type field. For an "A" record, this is a 4-byte IP address. For

Record Field Name	Description	Example Value
Name	Fully-qualified domain name of record	eecs.case.edu
Type	Type of record	A (IP-address)
Class	Allows for multiple independent namespaces; only one class in wide use	IN (Internet)
TTL	Time-to-live: the amount of time this record may be used or cached in seconds	86400
RDLength	Length of RData field	4 bytes
RData	Type-specific data	129.22.104.78

Table 2.1: DNS Resource Record Fields

an "MX" record, this contains the DNS hostname of a server responsible for receiving mail on behalf of a domain.

Chapter 3

Related Work

In this chapter, we discuss work related to our core topics of DNS, passive and active DNS measurement, distributed covert channels, peer-to-peer bootstrapping, and Internet naming and metadata storage systems.

3.1 DNS

As discussed in the introduction, before DNS, host naming on the Internet was accomplished via a centrally maintained and distributed hosts file [Kud74, HSF85]. As the number of hosts grew, manual maintenance of this file became onerous, and therefore the DNS was introduced. In 1983, the first specifications and implementation details of the DNS were introduced in [Moc83a, Moc83b]. The specification underwent major updates in 1987 [Moc87b, Moc87c], and numerous other Request-For-Comments (RFCs) have added to or refined the functionality of the DNS since.

3.1.1 DNS Measurement Studies

DNS measurement studies fall into two broad categories: passive studies, where logged DNS traffic from some vantage point(s) is analyzed, and active studies, where some set of

DNS actors is actively probed and evaluated.

Passive DNS studies allow researchers to examine the behavior of the DNS from one or more specific points of view for some time period. Passive measurements are useful as they provide a glimpse into the real-world operation of the DNS – whether from the view of some client population(s) or some authoritative server(s).

Active DNS studies generally allow researchers more flexibility than passive studies. For example, active studies can simultaneously measure performance from hundreds of different points of presence around the world, and can be tailored to target any service or use case desired. However, such studies are divorced from actual user traffic patterns.

For example, passively monitoring a few DNS resolvers would not assist in determining the prevalence of DNSSEC deployment on the Internet – as each resolver will fundamentally either implement it or not. On the other hand, active DNS measurements and probing cannot inform as to the cache hit rates experienced in the real world from geniune DNS traffic. We therefore conclude that both active and passive DNS measurements are crucial for gaining insight into the entire DNS ecosystem.

Passive DNS Measurement

We begin by discussing the passive analysis of DNS traffic. A study [MD88] by Mockapetris and Dunlap details experiences developing the DNS and makes some early observations of DNS traffic on the Internet. Conclusions from this work include the recognition of a need for negative caching and the observation that many resolvers would indiscriminantly cache data during lookups, leading to security vulnerabilities. Several years later, Danzig et al. perform a passive characterization [DOK92] of DNS traffic using two 24-hour traces monitoring a root DNS name server. This study concludes that the potential benefits of negative caching were dwarfed by the benefits that could be realized by fixing bugs in common resolvers – with bugs in recursion algorithms causing (potentially infinite) lookup loops. The authors conclude that fixing these bugs could decrease DNS traffic by as much as 20 times and outline some steps authoritative name servers can take to detect and log this buggy behavior. Throughout much of the mid-to-late 1990's, DNS research emphasizes security, discussed in Section 3.1.2.

Brownlee, et al. [BCN01] studies traffic appearing at a DNS root server, finding that over 14% of query volume at the root server was due to queries that violate the DNS specification – for example, that attempt to lookup an "A" record with a hostname that is an IP address. Jung et al. [JSBM02] analyzes both DNS and its followup traffic using traffic traces from two university populations. Important lessons from the Jung study include the observation that over 50% of DNS packets are queries or retried queries for lookups that go unanswered, an initial assessment of DNS client lookup latency on the order of 100ms, and a cache analysis that concludes that more than 10-20 clients will not meaningfully increase the hit rate of a shared DNS cache.

Several later studies examine query behavior at one or more root DNS servers. In [WF03, CWFC08], the authors assess the validity of requests arriving at one or more DNS root servers. Both studies find that between 2-3% of DNS queries are legitimate, while the remaining queries are sourced from misbehaving or poorly configured clients. These studies observe the most significant query types at the root include "A" record lookups for an IP address[1], queries for an invalid TLD, repeated DNS queries, and resolvers that fail to cache referrals from the root. Another study [LHF+07] examines root server traffic in the context of its anycast deployment, finding that instance selection via BGP is stable over the course of two days for 2-5% of clients.

Active DNS Studies

While passive DNS studies have been popular since the late 1980's and early 1990's, active DNS studies did not become prominent until the early 2000's. One early study is [WS00], in which the authors attempt to characterize the contribution of DNS to the total object

[1]Invalid as an IP address is the result of an "A" lookup.

retrieval time encountered while web browsing. In order to provide realism, the authors replay three days of web proxy logs into a live recursive resolver and study the resulting DNS transaction durations along with the already-logged web object retrieval times. The authors find that between 87-94% of web object retrievals use a DNS name that is already cached. The authors further find that 20% of the objects retrieved in a study of popular webpages experienced DNS lookup times of more than one second. Finally, the authors find that the DNS infrastructure for popular sites provides significantly faster lookups than for random sites – on the order of 2-4x.

A 2002 study [LSZ02] characterizes the difference in observed DNS performance for a variety of different users. The authors first crawl several popular websites recursively and compile a list of 15K DNS names. Next, the authors perform queries for each of these 15K names from each of 75 distinct vantage points (spanning 21 countries). The authors find a few characteristics that are consistent across vantage points, including the portion of responses that are CNAME aliases and the distribution of TTLs returned. However, performance related measures such as mean response time and the response times from root and TLD servers varied widely – the time to contact a root and/or TLD server varied from .037 seconds to 1.41 seconds depending upon the client.

A 2003 study [WMS03] analyzes the relative popularity of websites. In particular, the authors perform repeated queries for certain domain names to the local DNS cache, inferring domain name popularity from the amount of time that the domain name is in the cache. This work is important as it utilizes the DNS for a much different purpose than intended, as we do in Chapter 6. In fact, the results of [WMS03] could likely be significantly improved if the authors used the massive number of publicly-accessible DNS servers available on the Internet, as opposed to a small list of resolvers known to the authors. We leverage these widespread public DNS servers in both Chapter 4 and Chapter 6.

A later work [RMTP08] also attempts to infer website/service popularity through active probing of DNS caches. This work uses a larger list of 768K resolvers gleaned by retrieving the ADNS for popular websites that also provide recursive service (RDNS). In addition, [RMTP08] attempts to validate the results of the popular Alexa [Ale] ranking list of most-visited webites. The average distance from the position of a site in the Alexa list to the position of the site in the authors' list is 4.5, and that sites popular in only a specific geographic region showed the greatest discrepancy.

Another active study that infers data from DNS lookups is [SCKB06], by Su et al. in 2006. This study utilizes DNS responses from the Akamai [DMP$^+$02] CDN in order to derive information on Internet path quality, as Akamai is known to maintain a large number of servers and direct traffic among them based upon network conditions. The authors find that Akamai redirections strongly correlate with network conditions and that 70% of paths chosen by Akamai are among the best 10% of possible paths as measured by the authors.

In 2010, Ager et al. perform a study [AMSU10] of DNS resolvers, comparing those deployed within commercial ISPs to publicly-accessible DNS resolution services, such as Google Public DNS [gooa] and OpenDNS [Opeb]. The authors find that even with their distributed architectures, OpenDNS and Google DNS cannot achieve the kinds of low latencies observed for ISP-specific DNS servers. However, the authors also note that many ISPs use a load-balancing scheme that does not share a cache among resolvers, decreasing performance.

3.1.2 DNS Security

Early implementations of the DNS often lacked even basic security protections. From [MD88], "Several existing resolvers cache all information in responses without regard to its reasonableness." In his Master's thesis [Sch93], Christoph Schuba performs an analysis of the state of DNS security in the early 1990's. In particular, the author discusses how to compel a resolver to cache arbitrary data – which is used to demonstrate how to gain control of a system using the Berkeley "r-commands" (rlogin, etc) [RFR90]. This attack is also discussed in [Bel95]. The thesis presents several possible methods for hardening name servers and the DNS itself, including the concept of what would later be called "bailiwick checking"—where a DNS resolver only trusts the answers received from a name server authoritative for the zone containing the record. Bailiwick checking is discussed in-depth in [SS10]. The author also proposes a context cache, in which records would only be used in the context in which they were received. For instance, an "A" record received alongside an "NS" record to determine the location of a name server would only be used by DNS to contact that name server and not by clients resolving the name directly. In this manner, the ability of an attacker to overwrite a valid cache entry with a malicious entry is reduced. Another improvement suggested includes a cache bypass flag, by which hosts could request that certain sensitive DNS queries could bypass any intermediate caches. Finally, the author proposes using public-key cryptography to sign DNS messages for security.

Following the large number of attacks on the DNS, the community developed the Domain Name System Security Extensions (DNSSEC) [EK97, E$^+$99, AAL$^+$05] (among numerous other RFC's). DNSSEC strives to provide authentication to all DNS responses. In furtherance of this goal, each zone uses asymmetric encryption to digitally sign all of the records inside the zone, as well as the keys used by any subordinate zones. In this way, a DNSSEC-aware resolver can follow a chain of signatures back to the root (whose key fingerprints are well-known). One recent study [GC11] observed 8-10% of queries at the .org TLD come from DNSSEC-aware resolvers. The development and deployment of DNSSEC remains active within the community to this day.

In 2008, news of a severe flaw [Kam08b] in the DNS protocol itself was released. We discuss the details of this flaw in Section 4.6.1. The flaw allows an attacker to poison nameserver caches with arbitrary hostname-to-IP mappings within seconds in most cases. The attack, known as the "Kaminsky attack", along with variants and mitigations, is discussed in-depth in [SS10].

3.1.3 Communicating without Fixed Infrastructure

In Chapter 6, we introduce a novel communication technique requiring no fixed points of infrastructure. Our communication technique is comprised of two components: first, we enable two clients sharing some pre-determined secret (a string or application identifier) to discover a common set of open DNS resolvers on the Internet. Second, we leverage the caching and aging properties of the DNS resolvers discovered to store and read arbitrary data. While the channel we create is narrow, we have found it to be suitable for transmitting bootstrapping information (e.g., an IP address to use for further communication) or small text messages (on the order of a tweet, or 140 characters).

We are aware of little work closely related to the communication technique we present in Chapter 6. While our technique is not necessarily designed to be covert, it does utilize a covert channel [Lam73] of sorts. The existence of the particular channel we use was first mentioned in a Black Hat presentation [Kam04]. One slide of the talk described the RD method sketched in § 6.3. We are not aware of any empirical evaluation of the idea on that slide and therefore we conducted such an evaluation as part of [CAR12].

Additionally, while not directly comparable, our work shares some of the same goals as a number of other projects described in the literature. We sketch these below.

Collage [BFV10] is a system for circumventing censorship by leveraging Flickr as a communication channel. We share some of the same principles with Collage, though our goals differ. While Collage aims at being a part automatic, part manual method for rigorous censorship evasion, we aim to enable generic and programmatic communication without a central hub or human intervention.

Although the types of communication we enable are much different than those of an anonymizing network such as Tor [DMS04], we note several similarities in emphasis between the two systems. For example, network operators will be able to prevent the usage of both Tor and our system. Despite the fact that Tor can be and sometimes is blocked, its large user base suggests that network operators regularly choose otherwise. Therefore, we do not see this as major roadblock to our techniques either. Both Tor and our own system utilize intermediaries to pass messages from one peer to another, and though Tor takes explicit steps to provide anonymity for each node, in practice, obtaining the identities of actors in our scheme will be difficult as well. In either Tor or our own scheme, an ISP monitoring either end-user's traffic will be able to detect the covert communication and its first hop. However, cooperation of the first hop or its ISP would be required to reveal a second hop. As one of Tor's core purposes is anonymization, the second hop would likely be just another relay node, while in our system the second hop would be the other end-user. This "first hop" may be in a different jurisdiction, lack the necessary logs to tie actors together, or simply be uncooperative; this provides a degree of anonymity in our system.

A core use case of this system is peer-to-peer (P2P) bootstrapping; that is, the initial discovery of the IP address and port of a node within a P2P system by an uninitiated peer wishing to join the network. As our communication scheme does not require a pre-determined rendezvous address for communication, it is ideal for this application. We note some recent work in this area. For example, [DW09] uses random scanning to find members of the BitTorrent DHT, usually finding success after sending 30K–60K packets, which is two orders of magnitude more than our random scanning requires. In [DG08], the authors bias random scanning towards a prebuilt list of netblocks known to be rich in peers for a particular network. While effective for large, established networks, these methods are much more difficult for bootstrapping networks with only a handful of peers. Here, our advantage is in that we scan for open recursive DNS servers and use these to bootstrap, rather than scanning for the (possibly small) number of peer nodes directly. Furthermore, the random scanning schemes listed here assist only in finding *some* node of a target P2P network; our scheme allows any node to bootstrap efficiently with any other node sharing the same secret. In [WSJBF10], the authors propose using well-known P2P networks for bootstrapping smaller networks, and perform evaluations of several of these services.

3.1.4 Metadata Information Storage System

In this section we discuss work related to the Metadata Information Storage System, or *MISS*, discussed in Chapter 7. *MISS* is a foundational service that provides users and applications with a new *architectural abstraction* for dealing with meta-information. In the context of *MISS*, we define *meta-information* as the information needed by a user (names, social graphs, URLs) to get at the ultimate content they wish to interact with. The *MISS* system is motivated by the current mess of meta-information most users have accumulated—disparate application-specific configuration files, email addresses and URI's that have no contextual meaning to the user, "buddy lists" and other contact information that is typically bound to a single application, etc. In addition, we design *MISS* to open new possibilities for application design based upon easy sharing of data across applications, devices, and users.

The information contained in the *MISS* system is keyed to a *collection*, which is simply a group of records belonging to a particular user. Each *record* contains a name and a type, along with arbitrary application-generated data. The *MISS* system calls for each collection to be represented by a public key, and for records to be signed by the collection key. Records are optionally encrypted for privacy. Each user in the *MISS* system uses a *MISS Server*, which holds a public copy of the user's collection (encrypted subject to the user's privacy settings). Furthermore, each user has some device running a *MISS Daemon*, or *MISSD*, which provides an interface for local applications (not unlike a DNS resolver) and maintains the collection of records on the user's *MISS Server*. We will first discuss work related to the design and implementation of *MISS* itself, and then work related to some of the *MISS* use cases enumerated in Section 7.4.

Our use of cryptography to name principals and encode their relations was proposed in [RL96]. Our lookup model in which a local daemon (*missd*) resolves and fetches records from their servers is similar to the division of roles between DNS resolvers and authoritative DNS servers. We also simply re-use previously developed DHT technology in our current prototype, but nothing in our design is tied to a specific DHT implementation.[2] Another currently-implemented system that allows for arbitrary attributes to be associated with a (somewhat more weakly) secured identity is OpenID Attribute-Exchange (OAX) [opea]. Our system differs from OAX in that OAX is used to transfer a set of attributes to a remote server upon user login. If any of those attributes change, the remote server must be notified about the change—the remote server is unable to request attribute updates or additional attributes at will. Furthermore, OAX is not designed for user-to-user transfer of metadata, but rather user-to-service provider transfer. Finally, our system uses XML [BPSM$^+$00] for record encoding.

With respect to use cases, our notion of indirection—as used in the naming use case discussed in Section 7.4.1—can be viewed as similar to $i3$ [SAZ$^+$04]. The *MISS* naming use case is in part based upon the *personal namespaces* concept proposed in [All07]. Further, meta-information sharing resembles a number of proposals for Internet-wide user profile or identity management (e.g., [SHLX03, lib]). While these approaches focus on data models, *MISS* attempts to provide a neutral framework for applications to use however they see fit. The Windows Registry [win] is an example of a framework created for the purposes of application configuration, though it is tied to a single computer. Finally, the references given in § 7.4 as well as others (e.g., [TSGW09, TCL08]), show that in some cases similar tasks have been previously developed in an ad-hoc fashion. The contribution of the system we propose in Chapter 7 is not in any one of the technologies used or applications, but rather in the development of a pervasive *architectural abstraction* that we believe is useful for an increasing number of networked applications and services. We believe this new abstraction holds promise to both simplify current ad-hoc mechanisms and enable new services due to reducing the burden in dealing with meta-information.

[2] In fact, we interchangeably ran *MISS* on top of the Chord [cho] and Bamboo [Bam] DHTs.

Chapter 4

Client-Side DNS Structure, Security, and Protocol Compliance

In this chapter we analyze data collected from active measurements of the DNS from over 1 million vantage points across the Internet. We strive to study the DNS as it is experienced by clients in these locations. Therefore, we examine the aggregate DNS behavior from the combination of any ISP-level RDNS servers and/or a user's home gateway. We study the topology of client-side DNS resolvers, examine the caching and TTL behaviors experienced by clients, study the level of interference in responses by ISPs, and investigate deployment of a patch to the well-known "Kaminsky" vulnerability [Kam08b].

4.1 Client-side DNS Infrastructure

The architecture of the client-side DNS infrastructure varies across providers. The actors can be loosely grouped into three roles: (*i*) "ingress" servers that receive DNS queries directly from user devices, (*ii*) "egress" servers that directly communicate with external DNS infrastructure such as root and authoritative DNS servers, and (*iii*) hidden (from an external observer) servers that act as intermediaries between the ingress and egress. To avoid confusion, in the rest of this chapter we use the following terminology to describe the various components of the DNS infrastructure that is charged with obtaining a hostname to IP address binding from an authoritative DNS (ADNS). The actors are also illustrated in Figure 4.1[1].

- Origin devices are either user devices or the machines we use to probe the client-side DNS infrastructure.
- ODNS ("open DNS") are ingress DNS servers which accept DNS requests from arbitrary Internet hosts. This set includes both FDNS and RDNS$_d$ (see below).
- RDNS ("recursive DNS resolvers") are egress resolvers that communicate directly with authoritative DNS servers. This set includes both RDNS$_i$ and RDNS$_d$ (see below).
- FDNS ("forwarding ODNS") refers to an ODNS that does not itself lookup a submitted query, but rather passes the request along to another resolver.
- RDNS$_d$ ("direct RDNS") are RDNSs that are also ODNSs. In other words, an RDNS$_d$ is both an ingress and egress server.
- RDNS$_i$ ("indirect RDNS") are RDNSs observed at the authoritative DNS server resolving queries on behalf of an FDNS. RDNS$_i$ and RDNS$_d$ overlap; the intersection resolve queries on behalf of FDNS and origin devices.
- HDNS ("hidden DNS") are servers which operate between FDNS and RDNS. Since these servers are neither ingress nor egress servers, they are invisible externally. While we cannot directly observe these servers, their existence is confirmed by DNS operators [Goob] and therefore we must keep them in mind as their actions may impact our results.

[1] Figure courtesy of Kyle Schomp.

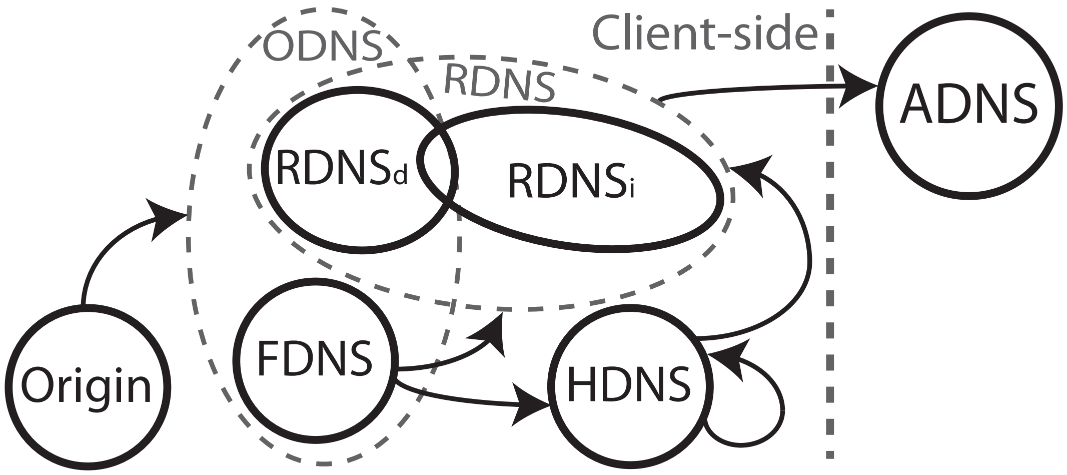


Figure 4.1: Structure of the client-side DNS infrastructure.

A typical example path through the maze of DNS-related devices has a client computer starting the process by sending a DNS request to a home routing device (FDNS). The FDNS in turn forwards the request through a chain of zero or more HDNSs and ultimately to an RDNS$_i$. The RDNS$_i$ sends the request to the appropriate ADNS. This general structure accounts for nearly all of our experimental observations.

4.2 Dataset

In order to drive this study, we scanned a large number of randomly chosen IP addresses by probing them with DNS queries for various subdomains within a domain name registered for the purpose [2]. The combination of probing open resolvers (ODNS) with running the authoritative DNS server (ADNS) that ultimately receives the queries enables us to discover and interact with resolvers that are otherwise not externally accessible on the Internet, e.g. resolvers operating only for the clients of a particular ISP. Our basic technique is an extension of the one used in [DPLL08a], which we extend to manipulate both the queries entering the resolving chain and the responses ultimately coming from our ADNS. We leverage approximately 100 PlanetLab [CCR$^+$03] nodes for scanning at a rate of two

[2]dnsresearch.us.

Criterion	No. ODNS	% ODNS
PBL Listed by SpamHaus	566K	51%
PBL Listed by ISP	180K	17%
Basic auth realm	265K	24%
RomPager	258K	24%
Wrong port	529K	48%
At least one of the above	849K	78%

Table 4.1: Home network device criteria

IP addresses per second per node for approximately two weeks—from February 29th to March 16th of 2012. Using this above scanning technique, we identify 1,093,583 ODNS willing to respond to our probes. We find ODNS in 208 countries and 10,075 Autonomous Systems (ASs). In addition, we discovered 55,497 $RNDS_i$ that initiated contact with our ADNS on behalf of some ODNS we probed.

ODNSs appear to be mostly home networking devices. We find that 78% are likely residential networking devices as they meet at least one of the following criteria as shown in Table 4.1: (i) HTTP probes to the device shows there to be a web server running that reports itself as RomPager, which is a well-known software package for creating web interfaces in embedded devices, (ii) HTTP probes to the device show the use of basic authentication with a realm indicating a likely home device (e.g., “DSL-2641R”), (iii) the IP address is listed in the Spamhaus PBL, or (iv) the device replies to a DNS probe from a port other than the port to which the probe was directed, which we speculate is indicative of a low-end Network Address Translation device (NAT) that is performing port translation on its own packets (these replies would normally be targeted at clients inside the NAT boundary).

4.3 Topology

We begin by looking into the topology of client-side DNS infrastructure. A majority of the ODNSs we discover contact our ADNS through at least one intermediate RDNS. Only 1.37% (around 15K) of the ODNSs contact our authoritative DNS server directly ($RDNS_d$) - the rest were FDNSs. In order to associate an FDNS with the RDNS(s) that it uses, we embed the IP address of the device we query in the DNS query itself. Therefore, our ADNS can view both the IP address actually resolving the query and the IP address to which the query was originally sent.

Further, in one phase of the scanning process the ADNS would embed the IP address of the RDNS from which a query arrives into the query response. When our probing node receives such a response, it attempts to directly contact that RDNS. Out of 44,393 such RDNS$_i$s[3], 16,789 or 38% accept our queries, while the rest fail to respond, refuse the query, or reply with an empty response.

Not only do most ODNSs utilize an RDNS in their DNS interactions, but they often utilize multiple RDNSs, which we call *RDNS pools*. We conclude that this fan-out behavior must be driven by ISPs and recursive DNS service providers, as consumer devices rarely allow for the configuration of more than handful of DNS server addresses.

We measure the sizes of these RDNS pools by sending multiple queries for unique hostnames to the same FDNS and observing the source IP addresses of the resulting queries arriving at our ADNS. The pool size estimates may in some cases be underestimates as they are limited by the total number of queries we send to each FDNS—the several tests we perform result in 282 queries sent to each FDNS. Still, Figure 4.2 reveals that most FDNSs utilize multiple RDNSs and a non-negligible number of them use rather large pools: over 30% of FDNSs use more than 10 RDNSs and 10% of FDNSs use over 30 RDNSs.

The level of functionality provided by FDNSs varies. For example, an FDNS might actually process the DNS queries in order to perform caching, or the FDNS might simply forward queries from clients to its designated RDNS "wholesale". We tested ODNS by issuing repeated requests for a name which would result in a random response on each query to our ADNS. We find that 40% of ODNS return two or more distinct answers—suggesting that these ODNS do not perform caching themselves. This figure is likely an underestimate, as an ODNS forwarding to a single RDNS will return a consistent result and

[3]This figure is smaller than the total number of RDNS$_i$ as only a portion of our queries returned the RDNS IP to the measuring node for reachability testing.

therefore not be detected by this test.

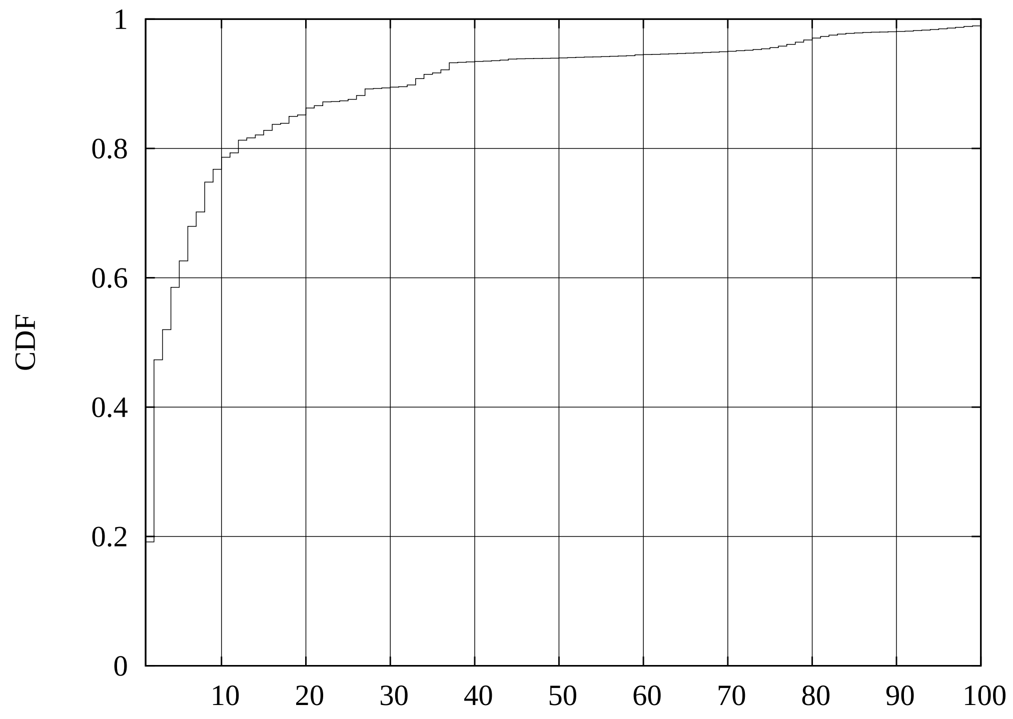


Figure 4.2: Pool Sizes (# RDNS seen for each ODNS)

Many Internet platforms—notably content delivery networks—rely on the assumption that client machines are close to their RDNS in that the platforms direct the client to a nearby node based on the location of the client's RDNS [DMP+02]. To test this assumption, we use a geolocation database [Maxa] to calculate the distance between FDNSs and the RDNSs they employ for resolution. Figure 4.3 presents the distribution of these distances[4]. As FDNS often use more than one RDNS, we plot two lines: the distribution of distances for each FDNS:RDNS pair and the distribution of distances for each FDNS to its most distant RDNS. We note that 85% of FDNS have all RDNS within 1000 miles—approximately 5 light-milliseconds. Nearly 30% of FDNS enjoy an RDNS situated so close that our GeoIP package cannot distinguish between then. We find the lines of Figure 4.3 cross at approximately 220 miles, suggesting that the FDNS using farther RDNS

[4]The accuracy of our result rests upon the accuracy of the geolocation database, which claims 81% accuracy within 25 miles for IP addresses inside the USA.

are also using more of them. We find that nearly 7% of FDNS appear to utilize RDNS that are at least 6000 miles away. Such FDNS would experience significant latency for DNS lookups. We note that this may not be intentional; for example, some malware has been known to change a user's DNS servers upon infection [FS]. Some of these cases may also be explained by analysis problems—the accuracy of GeoIP in general has been debated [PUK$^+$11, SZ11].

Another study has also considered the geographical distance between the client hosts and the DNS servers they employ [HMLG11] and found a shorter distance between clients and RDNS than in our experiment. The difference could be due to different vantage points. In contrast to their experiment which was driven by real user access to a given Web site, we actively scan for RDNS pools and strive to discover all RDNSs in a pool. We also note that we do not measure client location directly, but rather assume that clients are co-located with their FDNS.

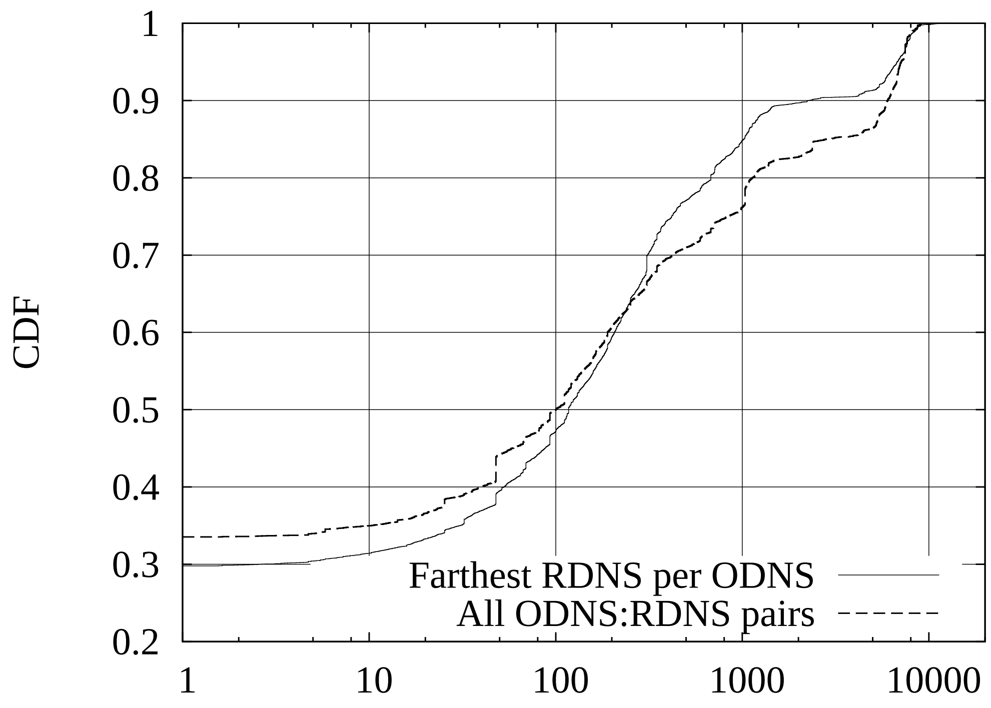


Figure 4.3: GEOIP distance from FDNS to RDNS (miles)

% of Servers Measured	Time Observed Alive
0.6	<= 10 min
2.2	(10min, 60min]
11.1	(60min, 9hr]
15.0	(9hr, 1day]
12.1	(1day, 3day]
58.1	Alive throughout study

Table 4.2: Time Spent Alive

4.4 Caching & Server Lifetime

We now consider the caching behavior of DNS resolvers. As our study of caching is constrained by the long-term availability of the ODNS server in question, we consider this issue first. As noted in [LL10a, DPLL08a], a majority of resolvers on the Internet are transient. In Table 4.2, we summarize the time ranges for which we were able to observe the ODNS resolvers in our study. As shown, a majority of servers remain active throughout our measurement period. While this finding might appear to deviate from [LL10a, DPLL08a], we note that we observe any particular ODNS for only three days, as opposed to the weeks used in these previous studies.

To study how long resolvers keep cached records, we send a series of probes to a given ODNS for a hostname in our domain that has a TTL of 1 million seconds, which exceeds the duration of our measurements. Furthermore, our ADNS returns a random answer for each one of these queries. By this manner, we can readily detect if we receive a cached response (identical to the previous one) or not. We increase the inter-probing time exponentially, with the first probe sent at time t=0 and successive probes sent at $t = 2^x$ seconds, with x incrementing from 1 to 18. We then track how long we can retrieve the record without another request being made to our ADNS. Figure 4.4 presents the complimentary cumulative distribution of this duration for each ODNS. We find that in 50% of cases where the ODNS stayed alive throughout the study, we are able to retrieve a cached copy of a record for 4.5 hours. Roughly 16% of servers that stayed alive would keep a record available for at least three days – the longest time we checked. Obviously, our ability to reliably retrieve a cached copy of a record decreases when we include ODNS that have ceased responding in the analysis. We note that some apparent evictions could be due to the use of an RDNS pool that does not share a cache. While some RDNS in the pool might contain a cached copy of the record, from the client's point of view an eviction has occurred, and therefore we count it as such.

We note that the exponentially increasing inter-probing time allows us to observe the amount of time that a record stays in the cache without being accessed at all. As caches commonly utilize a Least-Recently-Used, Least-Frequently-Used, or similar strategy for managing cache eviction, we capture the worst possible case for a record. We note that dividing the x-values of Figure 4.4 by two yields the amount of time a record stayed in the cache without having been accessed—in other words, a record that stayed in the cache until the three day point spent 1.5 days in the cache without being accessed. The fact that a majority of FDNS returned a cached record after at least 30 minutes without being accessed indicates a generally low level of cache contention. As at least 40% of FDNS do not cache locally, this result suggests many RDNS also have a low level of cache contention.

4.5 Record Management

In this section, we discuss modifications made to DNS records by devices on the user's DNS path, giving the user an unfaithful representation of the intent of the ADNS responsible for the record.

4.5.1 DNS Message Rewriting

In this section, we discuss instances where the mapping returned to the client is an overt lie. This behavior may be viewed as a security issue. We discuss two situations in which this can occur during normal operation.

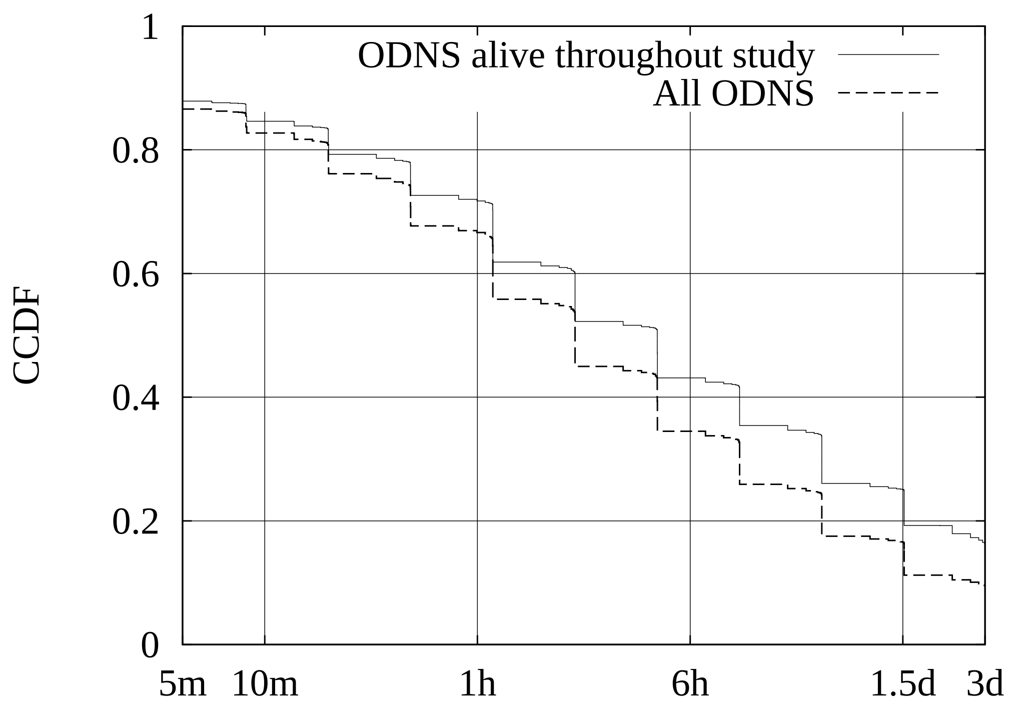


Figure 4.4: Duration of cached record accessibility

NXDOMAIN Rewriting

When a DNS request is made for a nonexistent DNS name, the authoritative server for that domain typically returns a message with a flag called “NXDOMAIN” set, indicating that no records exist for the given name. Furthermore, an “SOA” record is returned in the NXDOMAIN response indicating how long the NXDOMAIN response may be cached. However, many ISPs and DNS providers (e.g., OpenDNS) intercept these NXDOMAIN responses and rewrite them with a valid address – often directing clients to a landing page attempting to in some way monetize the user’s request. In our first NXDOMAIN test, we send a query for a specific name in our domain configured to return an NXDOMAIN response. This test yields 139,459 ODNSs experiencing NXDOMAIN rewriting. However this figure does not include OpenDNS (known to perform rewriting). Further investigation indicates we may have hit an edge case with our initial test – OpenDNS does not rewrite queries that appear to begin with an IP address[5] (as many of our test queries do). We performed a second NXDOMAIN test, requesting several non-existent top level domains – examining results from this query yields 259,458 ODNSs experiencing NXDOMAIN rewriting, a figure much closer to previous work [WKNP11]. Therefore, application developers cannot rely on honest NXDOMAIN behavior in the general case, however those still wishing to rely on NXDOMAIN behavior may increase their success by formulating DNS names as described above.

In order to attribute NXDOMAIN rewriting behavior to specific ISPs, we examine the set of RDNS involved in the affected resolutions. If a majority of the ODNSs served by a given RDNS return rewritten NXDOMAIN responses, we classify that RDNS as a probable NXDOMAIN rewriter. We then perform an ISP lookup on each RDNS using the MaxMind IP-to-ISP database [Maxb]. We found over 100 distinct ISPs/DNS providers we suspect to be performing NXDOMAIN rewriting. We list all such ISPs in Appendix A.

Search Engine Hijacking

In 2011, several ISPs were found to be hijacking client DNS requests for major search engines and responding with records containing proxy IPs in order to monetize client search traffic. These DNS aberrations were first described in [WKNP11, ZHR$^+$11], while the overall monetization strategy is described in [WKP11]. Given that our measurement techniques permit us to query many ISP's recursive DNS servers (even those hidden to the public), we search for this type of DNS rewriting. We found no widespread rewriting of DNS responses for these services. We contacted an author of one of the previous papers [Wea12], whose recent observations agree with these results.

[5]More specifically, OpenDNS appears to not employ response rewriting for queries of the form NNN.NNN.NNN.NNN.<any.thing>.tld, where NNN is 1-3 numeric digits.

4.5.2 Negative Answers

Our first test of DNS behavior with respect to nonexistent domains entails making a query to an ODNS for a domain that will return NXDOMAIN (a response code indicating the specified domain does not exist) only once, with repeat queries returning an "A" record response. In addition, the response to this first query contains no SOA record, and therefore should not be cached. We observe that 29% of ODNS have a resolution chain that performs multiple lookups on the first query, and therefore returns an answer from our ADNS. Contrary to specifications [And98a], 7.4% of ODNS cache the NXDOMAIN response – however the vast majority (over 98%) only cache this information for a matter of seconds. Only 846 ODNS cache this information for minutes or hours. We noticed a much higher rate of negative caching in a second test utilizing NXDOMAIN responses with SOA records as indicated in [And98a]. In this test, 57.7% of ODNS retain our NXDOMAIN response with associated SOA record. We observe 15.6% of ODNS were unwilling to cache our proper NXDOMAIN response; the remaining ODNS we are unable to classify due to packet loss or NXDOMAIN rewriting. We also note that 11.2% of ODNS return an NXDOMAIN for domains that do not exist, but strip off the SOA record that allows for that NXDOMAIN response to be properly cached. Furthermore, 0.3% of ODNS strip the NXDOMAIN error code, and return a DNS response with no answer.

4.5.3 TTL Treatment

We now turn to TTL treatment. While the TTL of a record given by an ADNS generally indicates how long that record may be used, it is crucial that administrators understand the circumstances in which these TTLs are disobeyed. In order to test TTL treatment we make a series of queries to each ODNS for records with TTLs varying from 0 to 1 billion seconds, increasing exponentially. It has been shown that clients often use cached DNS records longer than specified in the TTLs returned by authoritative servers [PAS+04a, SKAT12a]. Our measurement goes beyond this in that we consider whether resolvers obey the protocol

Expected (sec)	% Liars	Most Common Lie	% of Liars
0	11.43%	10,000	27.19%
10	11.1%	10,000	28.7%
100	2.96%	300	26.85%
1Ks	1.76%	80	30.07%
10K	2.85%	3,600	26.14%
100K	21.82%	86,400	52.6%
1M	89.35%	604,800	74.43%
10M	89.57%	604,800	74.16%
100M	89.58%	604,800	74.11%
1B	89.57%	604,800	74.12%

Table 4.3: Summary of TTL Deviations

in terms of the TTL values they return to the clients; in particular we find that resolvers often increase short TTL values and/or cap long TTLs.

We note that these two behaviors are not equally bad. When a resolver decreases a TTL, often the worst result is suboptimal DNS caching behavior. However, increasing a TTL opens the possibility that a hostname-to-IP mapping is used when it should not be. As with cache eviction, our experiment measures aggregate TTL treatment of the client-side DNS - including FDNS and any intermediate RDNS.

Short TTLs

Some applications, especially those involving DNS-based traffic control (such as those used by a CDN to direct clients to different replicas) may set a low TTL on returned records - on the order of tens of seconds. However, in practice these short TTLs are often disregarded.

To study resolvers' treatment of short TTLs, we configured our ADNS server to serve A records with short TTLs and then query ODNSs for these records. We find that approximately 21.6% of FDNSs return a record with the authoritative TTL of 0 seconds more than once without contacting our ADNS multiple times - hence from a cache. We note that records with a TTL of zero should not be cached for later use [Moc83b]. Furthermore, in 66% of these cases, a TTL other than 0 is returned to the probing node, whereas in the other roughly 34% the same record (with a TTL of 0) is returned multiple times. When we consider records with a TTL of 10, 11.6% of servers return the same answer after more than 10 seconds have elapsed, with 71.3% of these instances involving an overt lie regarding the TTL. In Table 4.3 we show that these lies can often be significant, increasing a record's purported validity by orders of magnitude.

Long TTLs

While the DNS specification [Moc87a] allows TTLs to extend to 4 billion seconds, in practice such high TTLs are not frequently permitted. To test resolvers' treatment of long TTLs, we request several records with large TTL values (up to one billion seconds). A majority (66%) of RDNS servers limit TTL values in their responses to clients to exactly one week. We find 11% will return a TTL of no longer than one day. Only 10% are willing to return TTLs of one billion. The remaining 13% utilize varying cutoffs.

Liars

As we show above, many resolvers will set both low and high boundaries for TTL values that they return to their clients. However there exists a large set of resolvers that lie about the true TTL of a record more arbitrarily. When we examine the set of servers that lie and return a TTL of 10,000, we find that in many cases they did so only for our records with TTLs of 0, 10, and 100, but would not rewrite records with a TTL of 1,000 or greater. Another class of device returns the correct TTL on the initial request, but subsequently serves constant, incorrect TTL values from the cache. In fact, we encounter only 8,445 (0.8%) servers that lie about the TTL for every request we send. Therefore, we conclude that many ODNS only reliably return TTLs in the range of minutes to hours. Finally, we identify 50,443 ODNS that return cached copies of the record without decrementing the TTL—leading to potential over-use of the records by clients.

4.6 Security

With the foundational role of DNS in today's Internet, DNS security has a profound affect on the security, trust, and operability of the overall Internet. A compromised DNS infrastructure that returns invalid name-to-address bindings has the general effect of diverting users to illegitimate services for various reasons. Possible nefarious behavior includes (i) replacing content, (ii) coaxing end hosts to run arbitrary javascript code, (iii) infecting hosts with drive-by malware and (iv) phishing users for credentials to legitimate web sites. In addition to changing DNS bindings, a compromised DNS infrastructure can also effectively prevent communication by refusing to resolve particular host names. In fact, we note DNS manipulation is a popular censorship technique [LWM07].

As Kaminsky discusses in [Kam08b], all that is required to obtain a legitimate SSL certificate for a given domain is the ability to receive email from the certificate provider for an arbitrary email address at that domain. If an attacker can poison the certificate provider's RDNS, they may simply rewrite mail exchange records for any domain – thus ensuring the receipt of email sent by the certificate provider to that domain and, consequently, obtain the domain's SSL certificate. Therefore, the SSL trust system largely relies upon the proper functioning of the DNS. These examples can go on.

The remainder of this section assesses the prevalence of two cache poisoning vulnerabilities, as well as the known mitigations.

4.6.1 Kaminsky's Attack

Kaminsky [Kam08b] describes a DNS cache poisoning attack which leverages the connectionless nature of typical UDP-based DNS requests to insert an NS record into the victim's cache. Recall that an NS record contains the hostname of the authoritative DNS server for all hostnames within a particular domain. For instance, Google has an NS record that indicates the authoritative source for the binding of "news.google.com" to an IP address.

Thus, injecting a bogus NS record mapping either “google.com” or “news.google.com” to a malicious ADNS would allow the attacker to divert client communication from “news.google.com” to a web server of the attacker’s choosing. The Kaminsky attack proceeds with the attacker A sending a large number of requests for hostnames within a domain to be poisoned, $P.com$, to a victim DNS server V with query strings of the form “random_string.P.com”. A legitimate response to such requests must (i) be from the ADNS for $P.com$, (ii) be directed to the correct UDP port number (the source port listed in the request message), (iii) contain the valid query string (quoted from the request) and (iv) use the transaction ID assigned in the transmitted request. However, A knows the query string and can readily determine and spoof the IP address of the ADNS—leaving only checks (ii) and (iii) as the protection against illegitimate responses. By sending a large number of different query strings, A can use brute force guessing of port numbers and transaction IDs until one response matches.

Mitigating the Kaminsky attack involves increasing the amount of entropy in DNS requests such that the cost of guessing is too high to successfully mount the attack. Increasing the entropy includes randomizing the DNS transaction ID—which Kaminsky showed was insufficient to prevent his attack when it was uncovered in 2007—and randomizing both the transaction ID and UDP source port—a more effective mitigation strategy. Additionally, so-called 0x20 encoding [DAV$^+$08] has been proposed, which involves the RDNS randomly changing the capitalization throughout query strings. The later works because authoritative servers are case insensitive in resolving the query but retain the capitalization in their response. This then adds to the unknowns about the response that an attacker is forced to guess—hence lowering the probability of generating a valid response. Since different query strings are different lengths, the amount of entropy added by 0x20 encoding is variable.

We assess the adoption of the strategies for increasing entropy within DNS requests by sending multiple requests for unique hostnames in our domain to each ODNS. These

	All RDNS		RDNS_i	
Observation	No.	Perc.	No.	Perc.
Static Source Port	9K	12.9%	4.6K	8.3%
Non-Random Query IDs	201	0.3%	66	0.1%
Uses 0x20 Encoding	195	0.3%	187	0.3%
Too few queries to tell	12K	17.6%	12K	22%
Total	69K	100%	55K	100%

Table 4.4: Kaminsky Security Observations (Note: the “0x20” line reports patched servers while the rest of the lines report unpatched servers.

requests are in turn filtered through RDNSs before ultimately arriving at our ADNS server. We examine successive queries from a single RDNS at our ADNS and log source port numbers, DNS transaction IDs, and check for case changes in the query string. Table 4.4 shows our results for both all RDNSs in our dataset and also for only the RDNS_i servers—which we know to be used by a population of clients. We find that roughly 13% of all RDNS and 8% of RDNS_i not only fail to randomize UDP source ports, but *use a static source port for all transactions*! Furthermore, we observe RDNS servers using static source ports in 37% of the ASs represented in our dataset, which illuminates the breadth of the issue. The table also shows that we observe only 0.3% of all RDNSs and 0.1% of RDNS_is fail to randomize DNS transaction IDs. Unfortunately, as noted earlier, Kaminsky [Kam08a] showed this is not enough to thwart the attack.

Turning to 0x20 encoding, we detect 0x20 encoding from 0.3% of the RDNSs and RDNS_is in our dataset. Interestingly, 129 of these RDNSs only changed capitalization in the three most-senior labels of the hostname thus underutilizing the full randomization potential of the queries. Additionally, our observations may well prove to be a lower bound on the adoption of 0x20 encoding. We are aware of at least one significant RDNS pool—Google Public DNS [Gooc]—that only employs 0x20 encoding on a white-listed set of domains. Our measurements therefore may be underestimating the amount of 0x20 encoding in proportion to the number of RDNS servers using such whitelisting. Unfortunately, we have no information on the prevalence of whitelisting.

Finally, we note that the rise of RDNS pools serves to mitigate the Kaminsky attack as well. Regardless of the IP address the attacker uses as entry point into the pool, the IP address used to communicate with the ADNS is chosen according to an algorithm unknown to the attacker. Therefore, to launch a Kaminsky attack against an RDNS pool, the attacker must either target every RDNS in the pool simultaneously or know how the pool distributes requests internally. For a perfectly random pool, the entropy of the DNS response the attacker must successfully spoof is increased by $log_2(N)$ bits, where N is the number of RDNS in the pool.

4.6.2 Bailiwick Rules Violations

Bailiwick rules are utilized by resolvers to prevent malicious ADNSs from inserting fraudulent records into the resolvers' cache [Ber]. The general form of this attack is that in a response from $X.com$ for some legitimate name—e.g., $www.X.com$—the "additional answers" section of the response could supply arbitrary unrelated bindings—e.g., for $www.Y.com$. To potentially save time later, a susceptible resolver adds both $www.X.com$ and $www.Y.com$ to its cache. During our "General" scan, we test for this vulnerability by sending a series of requests for hostnames from our domain and returning responses that included information for a non-existent google.com subdomain.[6] We then query the ODNS for this google.com subdomain and determine whether the response includes our poisoned result or an error message from Google indicating a non-existent domain.

Preventing this attack through the implementation of bailiwick rules—such as checking that any records in the "additional answers" section belong to the domain owned by the responding ADNS—is a basic security measure implemented in ordinary DNS servers. In our most simplistic attack, we find 675 cases where RDNS servers readily cache a DNS response for a hostname from the bogus google.com subdomain. Furthermore, we observe 231 cases where the resolvers would cache any additional record

[6]In other words, this will not interfere with regular Google traffic as the hostnames involved are not in use.

from a response to an Mail Exchanger (MX) query[7] (especially dangerous because it often enables hijacking SSL connections), and 203 cases where the resolver would cache any additional record from a Canonical Name (CNAME)-type query[8] Overall, there are a total of 749 cases where we find a resolver falling prey to at least one of these record injection attacks.

4.6.3 Additional Comments

Here we make two additional comments on DNS security that are informed by our measurements.

Laundering DNS Attacks

The prevalence of ODNS servers not only opens vectors for attack, but also allows for the ready laundering of DNS attacks. In other words, carefully attacking RDNS—say, using the Kaminsky attack or simple Denial of Service—via FDNS obscures the true source of the attack from the RDNS. Also, given the number of ODNS there is ample opportunity for attackers to "move around" to avoid being detected and/or to continue attacks even after a particular ODNS has been dealt with. The obvious way to address these issues is for home networking gear to stop answering DNS requests from the Internet-at-large and only honor such requests from internal hosts.

DNS Version Information

Servers that provide version information for the asking are not necessarily any more vulnerable to attack than servers that do not. However, providing such information does in fact lower the bar for attackers in that it can help attackers winnow the attack-space to only

[7] An MX record designates a hostname responsible for mail operation for a domain; such responses may include the IP address for that hostname as an additional record.

[8] A CNAME response indicates that the queried hostname should be replaced by a given hostname and the lookup process continued. As with the MX record, the IP address of the new hostname may be provided as an additional record.

Version	Count
Bind 9	95K
Nominum Vantio	64K
dnsmasq	36K
PowerDNS	17K
Bind 10	12K
Obscure or unconfirmed versions	11K
Bind 8	2K
Non-reporting	860K

Table 4.5: Observed DNS Resolver Versions

those vectors to which the given implementation is vulnerable. During our scan we request "version.bind" information from all 1.09M ODNSs we encounter. We cannot, however, tell whether the returned information is from the FDNS, RDNS or HDNS. The point is to assess how often information is being leaked that could potentially aid an adversary. Table 4.5 lists the breakdown of the responses. Due to the prevalence of Bind DNS servers, we split them by major version number; all versions of other DNS server implementations are grouped together. We find a number of popular DNS server implementations are guilty of releasing version information. The "non-reporting" servers either return empty responses, strings unrecognizable as versions, or attempts at humor, such as "My name is Bind, James BIND". Overall, roughly 22% of the resolvers in our dataset are willing to disclose their version.

Chapter 5

A Glimpse into DNS Usage from a Traffic Engineering Perspective

In this chapter we describe an initial study of modern DNS behavior as observed from the vantage point of clients within a small residential network over a recent 14 month span. While some of our analysis is a reappraisal of previous work conducted by others [JSBM02, PAS$^+$04b], we are aware of no recent analysis that passively assesses DNS behavior even though the DNS ecosystem is constantly evolving—e.g., due to increasing use of open global DNS platforms, the popularity of DNS pre-fetching, the increasingly complex structuring of the recursive lookup process that involves pools and hierarchies of servers as discussed in Chapter 4. This chapter proceeds in four steps: (i) in § 5.2 we study client behavior as reflected in DNS requests, (ii) in § 5.3 we turn to the ADNS server behavior that manifests in DNS responses, (iii) in § 5.4 we briefly discuss DNS transmission characteristics and finally (iv) in § 5.5 we turn to investigating the use of resolved hostnames by clients.

5.1 Datasets and Methodology

For this study we monitor DNS traffic within the "Case Connection Zone" [ccz], which is an experimental network that connects roughly 90 homes in a neighborhood adjacent to Case Western Reserve University to the Internet via bi-directional 1 Gbps fiber links. The connections from each house come together in a switch. We have a packet-level monitor that receives all traffic via a mirroring port on the switch. The CCZ network is atypical in terms of the capacity afforded to users. However, a companion study [SSDA12] finds that the usage patterns are fairly consistent with those other residential network studies have shown. It further finds the average CCZ user exceeds 10 Mbps for 1.2 minutes per day. This suggests that CCZ users' behavior is not to a first order driven by their ultra-broadband links.

Our measurement apparatus drops on the order of 0.01% of the packets in the worst case [SSDA12]. At this measurement-based loss rate we do not believe the insights from our analysis are skewed by our apparatus. Additionally, we experience measurement outages where no traffic is observed. These are caused by mundane logistical failures (e.g., temporarily unplugging the monitor from the network or disks filling) and from what we can tell not by network phenomena that would suggest a biasing of the resulting data. Our vantage point lies between the users' devices and the recursive DNS server provided for the users and therefore we directly observe client behavior not filtered by the recursive DNS servers.

We use the protocol analyzers found in the Bro intrusion detection system [Pax99] to reconstruct DNS queries and responses from the observed packet stream and log these to text-based logs via Bro's standard `dns.bro` policy. The DNS logs include timestamps, involved IP addresses, error code, contents of the question and answer sections of DNS requests and responses, and a summary of the number of records in the authority and additional answers sections of DNS responses. Some of our analysis aims to understand traffic resulting from DNS lookups. For this we use Bro's standard connection log—as produced by the `conn.bro` policy—which summarizes each transport layer connection and includes timestamps, durations, involved IP addresses/ports, amounts of data transferred, application protocols, as well as Bro's standard notion of the connection's ending state (e.g., "normally terminated") and history of key actions observed during the connection (e.g., three-way handshake completed, TCP reset observed, etc.).

The dataset used in this work was collected continuously from January 25, 2011 through March 31, 2012. During this period we collect just over 200 million DNS queries and about 162 million DNS responses that yield an IPv4 address. These queries were made by roughly 85 client IP addresses.[1] In addition, the connection logs contain 1.1 billion flows over the entire dataset. Of the connections we link with preceding DNS traffic we find 92.2% are HTTP—including ports 80 and 443. This is not surprising since the bulk of the connections initiated by CCZ users are either web connections or peer-to-peer connections [SSDA12], and peer-to-peer systems generally use DNS sparingly. For instance, while a BitTorrent client may use the DNS to access a tracker web site to obtain a description of a torrent of interest to the user, the IP addresses of the peers in the given swarm are given by the tracker[2] without relying on the DNS.

5.2 DNS Requests

Communication across the Internet generally starts with a DNS query from a user-facing device. Some applications continue to heavily rely on DNS over the course of their operation (e.g., web browsers), while others only use DNS for bootstrapping (e.g., peer-to-peer applications, as discussed above). Fundamentally the request to resolve a hostname into an IP address is straightforward and the DNS protocol is likewise uncomplicated. In subsequent sections we will show that the complexity of the system increases when we start to

[1]Note, the CCZ project provides a router to each house and this device NATs internal hosts. Therefore, while we observe 85 client IP addresses, this represents more than 85 users. See [SSDA12] for a deeper analysis of the number of users.

[2]Peers within the swarm can identify additional peers by IP address after the client joins the swarm.

Req. Type	5^{th} perc.	Median	95^{th} perc.
A	76.7%	87.5%	90.0%
PTR	5.3%	8.7%	16.9%
AAAA	2.4%	4.1%	9.7%
OTHER	0.0%	0.0%	0.2%

Table 5.1: Queries by type per month.

study how these simple requests are handled by the DNS ecosystem and then in turn how the responses are dealt with by the end hosts. The results in this section are basic context for the remainder of the chapter.

Table 5.1 shows the most prevalent DNS query types in our dataset. The table shows that type A queries for IPv4 addresses associated with some hostname are the most prevalent with over 87% of the requests per month at the median. The PTR lookups appear to be predominantly Bonjour discovery traffic. We also find that AAAA lookups for IPv6 addresses constitute a median of 4.1% of the queries per month even though we find almost no actual IPv6 use by CCZ hosts. We believe this is caused by some operating systems making A and AAAA queries simultaneously regardless of whether or not the host intends to use IPv6.

We next turn to understanding the breadth of DNS hostnames requested by the CCZ users. Figure 5.1 shows the distribution of popularity by hostname and by registered second-level domains (SLD)[3] over our entire dataset. We find heavy-tailed distributions with approximately 63% of hostnames and 50% of domains requested only once over the entire 14 month measurement period. However, we find millions of requests for the most popular hostnames and domains. We also note that some domains use a large number of subdomains—we observe over 100K distinct subdomains (in this case, third-level domains) for each of google.com, blogspot.com, and tumblr.com.

Finally, we note that roughly 97% of the DNS requests we observe traverse one of the two local resolvers provided for CCZ clients. The primary resolver is used for

[3]While technically second-level domains include only the two most senior labels in a hostname (e.g., “google.com”), we also include a third component in our analysis if that third component is what an organization buys from a DNS registrar (e.g., “theregister.co.uk”).

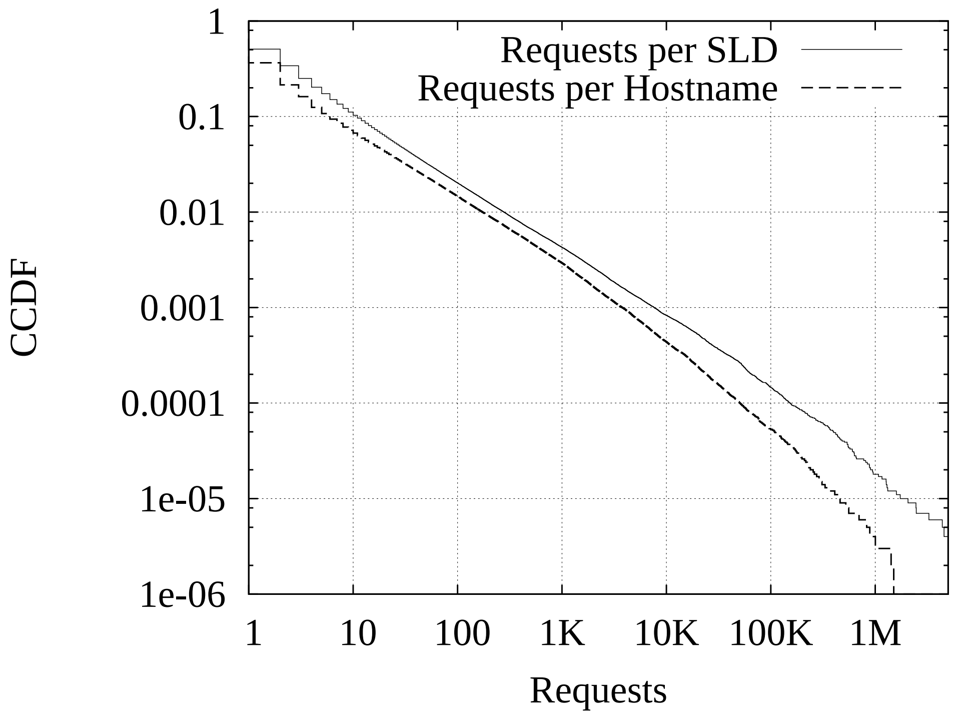


Figure 5.1: Distribution of # of requests per hostname and SLD.

approximately 81% of the overall requests and the secondary resolver handles 16%. These resolvers are not specifically for CCZ users, but are more broadly used within the ISP providing access to the CCZ network (OneCommunity). Google's public DNS resolver handles just over 1% of the requests we observe. The remaining 2% of the requests are dealt with by myriad resolvers, including Level3's open resolver, OpenDNS and a Case Western Reserve University resolver.

5.3 DNS Responses

We next analyze various aspects of DNS responses to understand how content providers leverage the DNS to direct traffic.

5.3.1 TTLs

We first study the time-to-live (TTL) associated with hostnames in DNS responses. The TTL is assigned by the authoritative DNS server and indicates the time period for which the response is valid. The TTL aims to provide consumers with the ability to cache the result of a lookup such that future requests for the same name can be handled locally, while at the same time allowing the name-to-address associations to expire at some future point such that they can be changed. Longer TTL values allow for higher cache hit rates (and therefore decreased DNS traffic), while lower TTL values allow a service provider more agility in terms of changing their mappings and directing traffic. Precisely where to place the TTL is a complex engineering tradeoff that is ultimately a policy decision for operators.

Most of the user-facing devices within the CCZ use a local recursive DNS server to traverse the DNS hierarchy and find hostname-to-IP address mappings. Therefore, responses can be served from the cache of the recursive resolver and hence the TTL can be less than that originally assigned by the authoritative nameserver. We seek to understand the TTLs being assigned by the authoritative servers. While our dataset does not contain direct interactions with authoritative nameservers, the recursive resolvers looking up a name for the first time should return the TTL assigned by the authoritative server. The recursive resolver will return smaller TTLs for subsequent lookups. Hence, we use the maximum observed TTL for each name as an approximation of the value assigned by the authoritative server.

In Chapter 4 we observed that some resolvers return bogus TTLs or incorrectly decrement the TTL when responding from the local cache. We therefore tested the TTL handling of the resolvers monitored in this study to ensure our maximum TTL approximation is reasonable. We sent the two CCZ resolvers queries for hostnames from our own domain. When these queries arrive at our authoritative DNS server we respond with a record containing TTL values ranging from 10 to 1M seconds. We then probe for the same names repeatedly to assess the resolvers' TTL decrementing behavior. We find that both monitored resolvers report correct TTLs for values up to one week. TTLs over one week are set to one week. Further, both servers correctly decrement TTLs when responding from the cache.

Figure 5.2 shows the distribution of the maximum TTL for each hostname in our dataset. The figure shows multiple modes that indicate popular TTL values—e.g., 20 seconds, 60 seconds, 1 hour, etc. We find a median TTL of 5 minutes and that only roughly 1% of the TTLs exceed one day. We also find roughly 40% of the hostnames have TTLs of at most 1 minute—which is suggestive of fine-grain traffic engineering.[4] We also analyze TTLs across time in our dataset and we find that the 5^{th} percentile, median and 95^{th} percentile per month are relatively stable across the 14 month dataset at roughly 5–20 seconds, 300 seconds and 1 day, respectively.

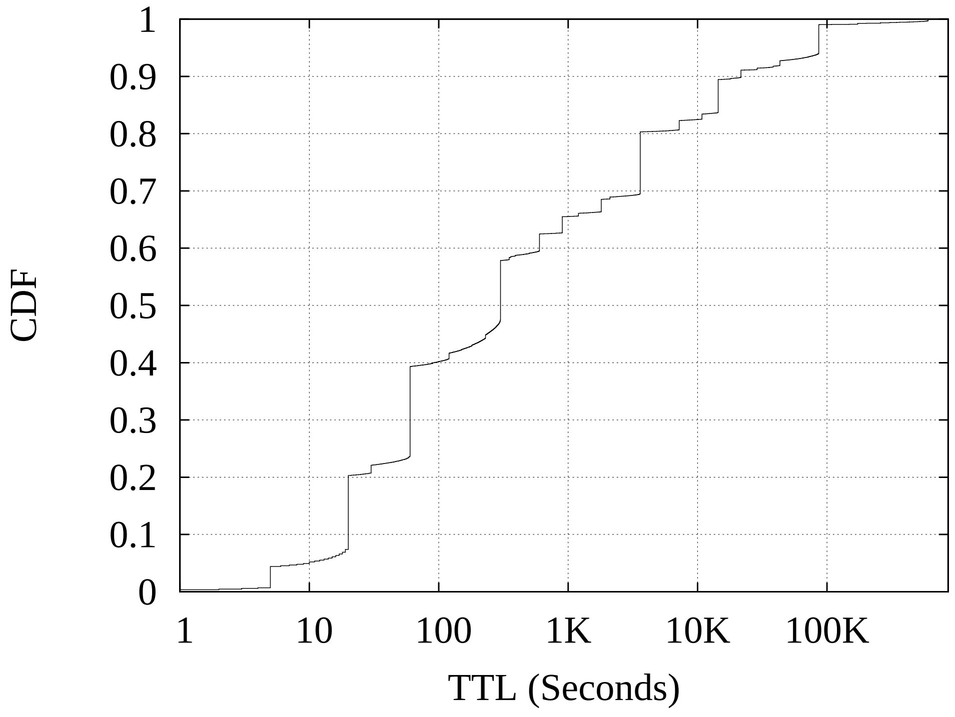


Figure 5.2: Max. TTL for each distinct answer

[4]We have no insight into the reasoning behind the TTL values in various responses. The values ultimately come from policy decisions encoded by authoritative nameservers. However, these low values seem likely used to enable flexibility in terms of directing traffic as many of these are from well-known CDNs (e.g., Akamai uses TTLs of 20 seconds).

5.3.2 Equivalent Answers

Individual DNS responses may contain multiple mappings in response to a single query for two reasons. First, multiple IP addresses give clients recourse if they cannot communicate with a particular IP address. Second, this aids load balancing across a set of replicas as recursive DNS servers will rotate the order of the IP addresses returned when serving records from the cache.[5] Figure 5.3 shows the distribution of (i) the total number of IP addresses returned for each hostname across all responses in the entire dataset, (ii) the average number of IP addresses returned in each individual response for each hostname in the dataset and (iii) the average number of nameservers (NS records) returned in each response for each hostname in the dataset.

We find that roughly 75% of the hostnames map to a single IP address over all requests in the dataset, with another roughly 8% mapping to two IP addresses. Further, we find that over 11% of the hostnames associate with at least five IP addresses over the course of our dataset. So, while traffic engineering may not be in play for most hostnames resolved within our dataset, some hostnames clearly use a variety of replicas. In terms of the average number of IPs in each DNS response, we find that 79% of hostnames return a single IP address per response (though in some of these cases, the IP address returned differs from response to response).

The final line on the plot shows the distribution of the average number of nameservers identified for each hostname. We often find more nameserver records in a response than IP addresses. Since NS records form the basis of queries for IP addresses of content servers we hypothesize that providers work hard to ensure resolvers have a reliable and up-to-date set of NS records. While only two nameservers are required to register a domain, almost 50% of responses include at least four NS records in their responses—underscoring the crucial nature of these records.

[5]We are not aware of this behavior specified anywhere. However, [AL01] discusses the behavior in terms of the popular DNS server *bind*. Further, we do observe this behavior from the predominant local resolvers used by CCZ hosts.

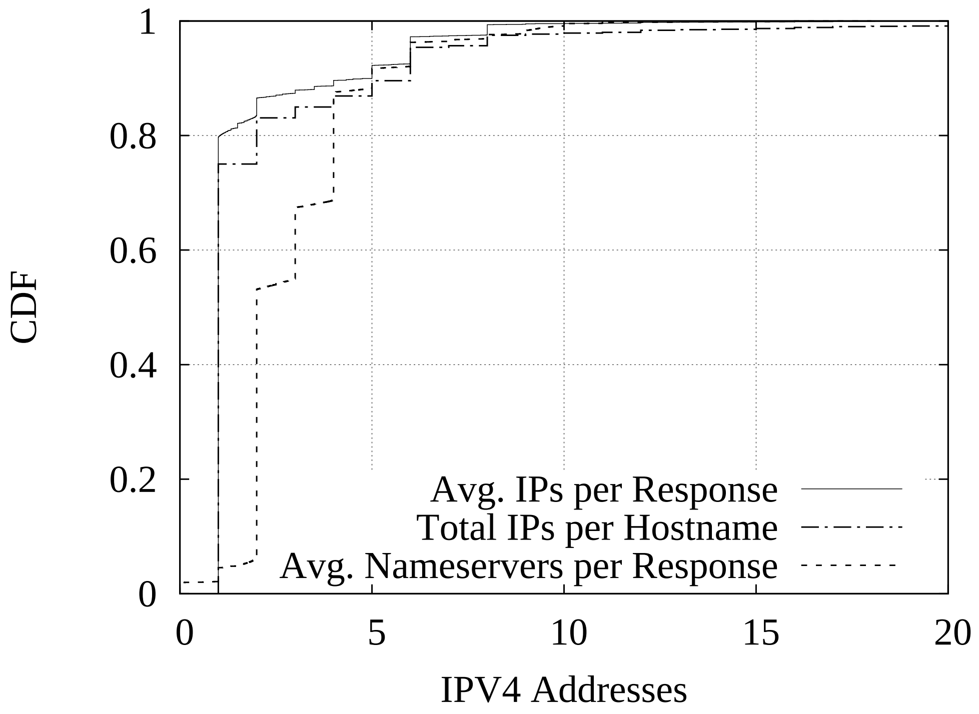


Figure 5.3: A & NS Record Distributions

5.3.3 Proximity

We next turn to understanding the geographic distance between returned IP addresses and the requesting user device. It is well known that popular services attempt to provide content from nearby servers and that directing traffic via DNS is a common way to drive clients to close servers (e.g., this is what the Akamai CDN attempts to do [DMP$^+$02]). We use a geolocation database from MaxMind to map IP addresses to locations at a city-level granularity [Maxa].[6] Since DNS responses can have multiple answers, we consider the geographic distance to each position in the DNS response independently. For all DNS responses, we calculate the average distance to each position in the response. For all such averages we then compute the quartiles for each position and each DNS response size. For instance, for responses with two IP addresses we calculate the quartiles for the first and second positions independently. We determine the quartiles for DNS responses containing

[6]As our study spans 14 months and GeoIP data is not static, we use archived GeoIP databases corresponding to the month of each DNS transaction.

# IPs	25^{th} perc.	Median	75^{th} perc.
1	851	2,036	2,159
2	851	851	1,741
3	851	1,119	2,159
4	2,154	2,154	2,156

Table 5.2: Distances (miles) to returned IPs.

1–4 IP addresses. We find the distance quartiles are the same regardless of position in the DNS response. This underscores the equivalence of these addresses, as discussed above. There is no proximity preference in the ordering of answers in a response. As we will show below, providers typically attempt to list a set of equally close addresses.

Given that the quartiles for the positions within responses of a given size are the same, we ignore position and report the distance as a function of the number of IP addresses in a response in table 5.2. We find that hostnames that map to only a single IP address are less popular and hence not optimized for locality and are roughly 2K miles away from the CCZ clients at the median point. Hostnames that use two or three IPs per response are the closest to the clients. This follows our intuition given that Akamai is heavily used for CCZ traffic [SSDA12], returns two IPs per response [SCKB06] and is known to use geography as a criteria in choosing replicas for clients [DMP$^+$02]. Finally, we note that DNS responses listing four IP addresses are generally further away from the clients than those with fewer addresses. We have no ready explanation for this observation.

Since we find the distance between the CCZ and the location of the IP addresses provided in DNS responses to be invariant of the position within the response we hypothesize that the IPs in a given response are generally within the same geographic area. We test this hypothesis by computing the average of the pairwise geographic distances between each pair of IPs in each DNS response. We find the average pairwise distance to be less than 10 miles in over 93% of the cases—showing that indeed the IPs returned as part of a given response are largely within the same location.

5.3.4 Case Studies

As discussed above, the modern DNS ecosystem is widely used for traffic engineering and replica selection as opposed to simple hostname lookups. The subsections above provide information concerning this traffic engineering process along a number of dimensions, but these do not fully capture the dynamic nature of the DNS resolution process. While we do not yet have a comprehensive model for capturing highly dynamic DNS traffic engineering activity, it is a topic of future work. As a glimpse into the process we briefly study the replica selections of two popular CDNs: Akamai and Google. We should stress that our discussion here is in terms of the IP addresses returned in DNS resolutions; because of a possibility of multiple addresses assigned to the same physical machine, or addresses belonging to virtual rather than physical machines, or dynamic IP addresses, we cannot exclude a possibility that the distinct IP addresses may not correspond to distinct physical replicas.

We first look at the answers returned by Akamai on behalf of iTunes. Previous work shows that Akamai uses a large number of servers [DMP$^+$02] and performs fine-grained replica selection based upon network conditions and load [SCKB06]. Figure 5.4 shows a scatter plot where each point represents an IP address from a DNS response. The x-axis shows the time of the response and the y-axis shows the IP address—where the addresses have been mapped to integers based on the order of first observation. We observe 140K responses during the course of our dataset and find nearly 300 server IP addresses. We find a general base set of addresses across most of our observation period as shown by the two thick black lines in the plot. However, we also find many additional servers throughout our measurement period. As we will show in the next section, clients use the first IP address in DNS responses with higher likelihood than subsequent addresses. We constructed a plot similar to figure 5.4 but considering only the first IP in each response. We elide this plot as it is largely indistinguishable from figure 5.4. However, the close similarity between the two plots indicates that the IP address rotation performed by the recursive DNS server provided for the CCZ users is in fact working to distribute load across the advertised servers.

A second case we examine is the replica selection over time by the Google CDN serving YouTube content. Although YouTube employs a complex combination of HTTP redirections and DNS resolutions to distribute user requests [AJCZ12], we focus purely on DNS resolutions. Figure 5.5 shows the timeline of nearly 40K responses to CCZ clients. While Google maintains its own CDN, we do not observe the variety of redirection as we do with Akamai. We find just over half as many server addresses in use for YouTube as we find for iTunes/Akamai. Further, while we find no base set of servers in use throughout our dataset, the responses are more stable across time than those of Akamai. We do find sudden shifts in server addresses at various points in the dataset, but server assignment between shifts is more orderly than for the iTunes replicas shown above.

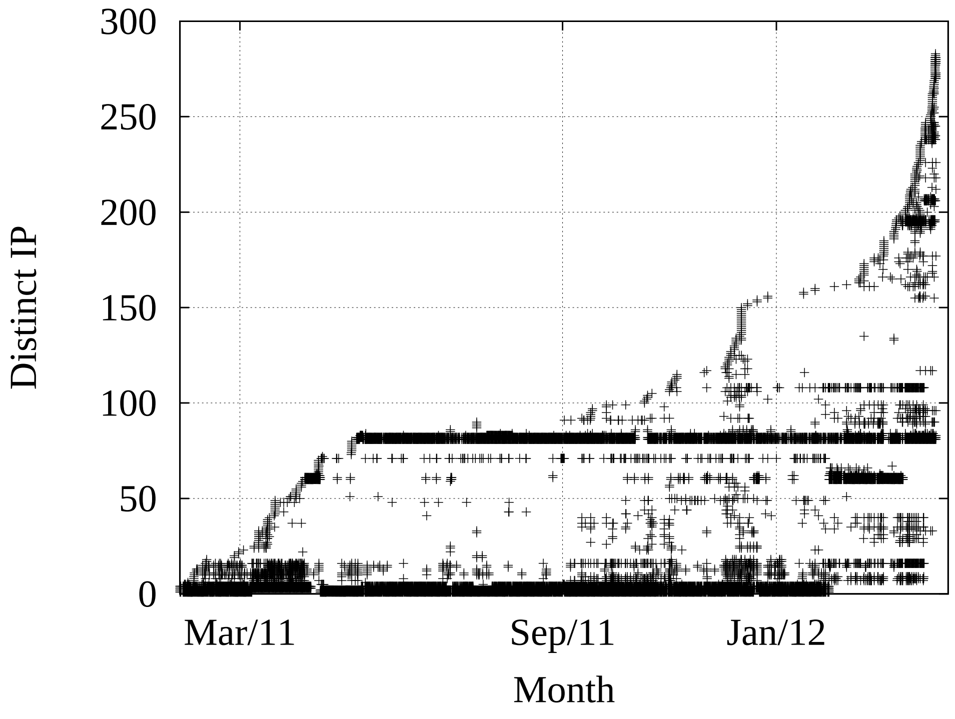


Figure 5.4: Akamai IPs Returned Over Time

These two cases are illustrative and not necessarily indicative of any general patterns we might find, but rather show that different strategies for leveraging the DNS to direct clients to replicas are in use by different providers. By considering request routing from a

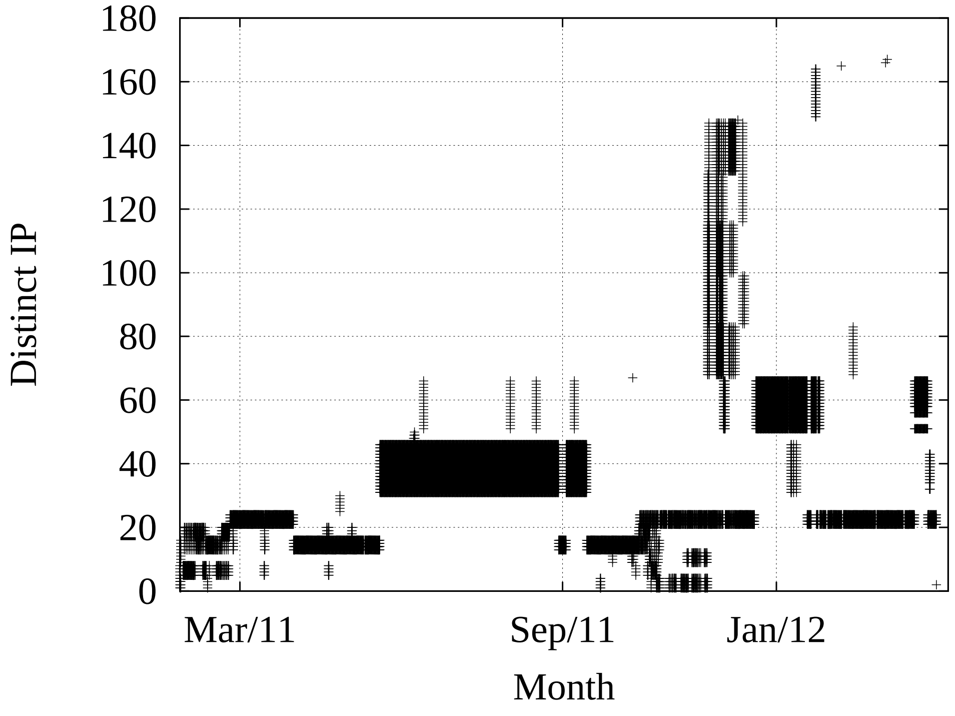


Figure 5.5: YouTube IPs Returned Over Time

client site's perspective, our observations complement studies aimed at mapping the content delivery infrastructures as a whole such as [AMSU11, TWR11].

5.4 Transmission

While we aim to understand various facets of clients and servers via DNS requests and responses above, we now turn to the process of transmitting DNS mappings. The first order property of DNS interaction is speed and therefore we analyze the response time of DNS lookups, i.e., the time between transmitting a DNS request and receiving its response. For this analysis we first remove the 3% of the transactions in the dataset that do not involve the CCZ recursive resolvers (as discussed in § 5.2). Using an alternate resolver makes the measured response times not directly comparable with those transactions using the CCZ-provided resolvers. To make our analysis tractable we then uniformly sample 10% of the transactions from across the dataset to further analyze.[7]

Figure 5.6 shows the distribution of the response times of the DNS lookups. Our first observation is that roughly two-thirds of the transactions complete in under 1 msec.[8] This suggests that these DNS requests are being serviced from the cache at the CCZ resolver. Another roughly 10% of the transactions take approximately 10 msec. These transactions are likely associated with traffic to nearby institutions (e.g., traffic with the Case network or one of the nearby hospitals). The third region of the plot shows that the longest 25% of the transactions are well spread across the interval between 10 msec and 1 second. These are transactions that have to make their way through the global DNS ecosystem to a variety of authoritative DNS servers around the world.

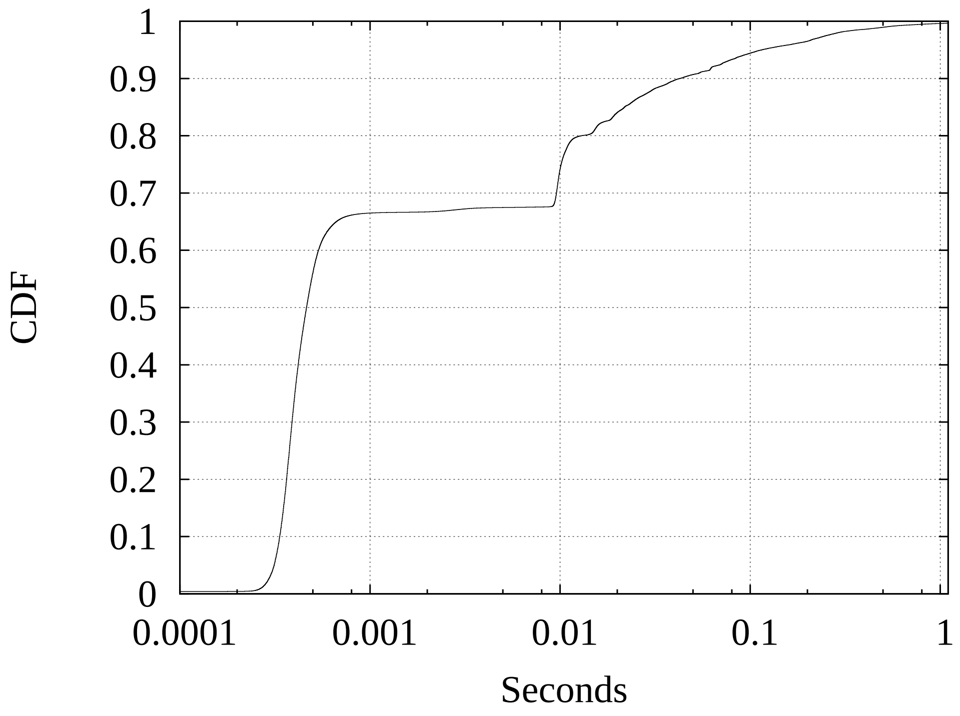


Figure 5.6: All DNS Response Times (10% sample)

Figure 5.7 shows the DNS response time for lookups that are not served from the

[7]We tried different sampling rates and the results hold and therefore we do not believe the sampling introduces bias into the analysis.

[8]Recall that our vantage point is at the aggregation switch, and thus our observed durations do not include the delay between the switch and the client, which we found to also be on the order of a millisecond. However, for purposes of identifying behavioral regions, the absolute value of the duration is not important.

CCZ resolver cache (i.e., the tail of figure 5.6). We find the median response time of a lookup not utilizing the cache is about 20 msec. While this is in fact lower than previous studies report [JSBM02, AMSU10], we are not able to determine any sort of concrete trends due to the difference in vantage points involved. We also compute the minimum DNS response time to each SLD in our dataset. The distribution of these minimums is also shown in figure 5.7. We find that the minimum per-SLD distribution shows longer lookup times than when considering all transactions. This indicates that the overall distribution of lookups is skewed towards closer DNS servers. Or, put differently, popular hostnames are served from closer authoritative servers than less popular hostnames. In fact, we find that 23.5% of DNS responses include a redirection to an Akamai server and 13.4% to a Google server. Given these well connected heavy hitters, it is unsurprising that the overall response times are skewed to lower values.

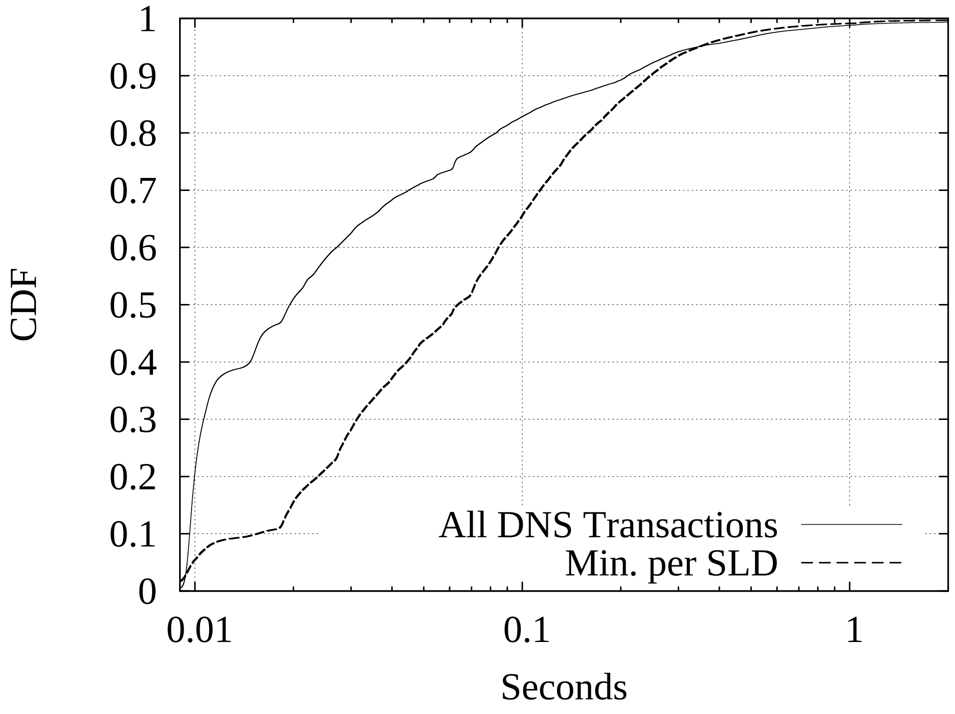


Figure 5.7: Uncached DNS Response Times

5.5 Utilizing Responses

We now tackle questions pertaining to how user-facing clients employ the information conveyed in DNS responses.

5.5.1 Use of DNS Responses

The first aspect of clients' reaction to DNS responses is simply whether or not the responses are used for subsequent communication. We find that across our entire dataset only about 60% of the DNS responses contain IP addresses that are used in any way for follow-on traffic. Another CCZ study uses OS fingerprinting techniques to roughly quantify the number of users behind the NATs in the CCZ [SSDA12]. That work also determined that users within the CCZ make heavy use of the Chrome web browser [Sta12], which employs DNS prefetching to reduce the web page load time experienced by users [chr]. While we cannot distinguish between DNS prefetching and other behavior that may cause DNS responses to go unused, it seems likely that DNS prefetching is at least partially responsible for the 40% of responses that go unused.

Given that we suspect DNS prefetching is in use, we next examine the time between a DNS response arriving and subsequent traffic involving an IP address included in the DNS response for for the 60% of the DNS responses with follow-on traffic. Figure 5.8 shows the distribution of the elapsed time between the DNS response and connection initiation. We find that connections are initiated within 100 msec roughly 79% of the time and within 1 second roughly 87% of the time. These are likely cases where the application is directly using the DNS responses it receives and not trying to optimize future communication by prefetching mappings that may or may not be needed. Therefore, of the DNS lookups that are ultimately used, at most 13% are the result of prefetching. Without additional information about application logic this bound may be a tenuous presumption at the lower delays. However, we note that in 4.6% of the cases the use of an IP comes at least 1 minute after the DNS lookup. Such delays are difficult to attribute to anything other than speculative DNS lookups.

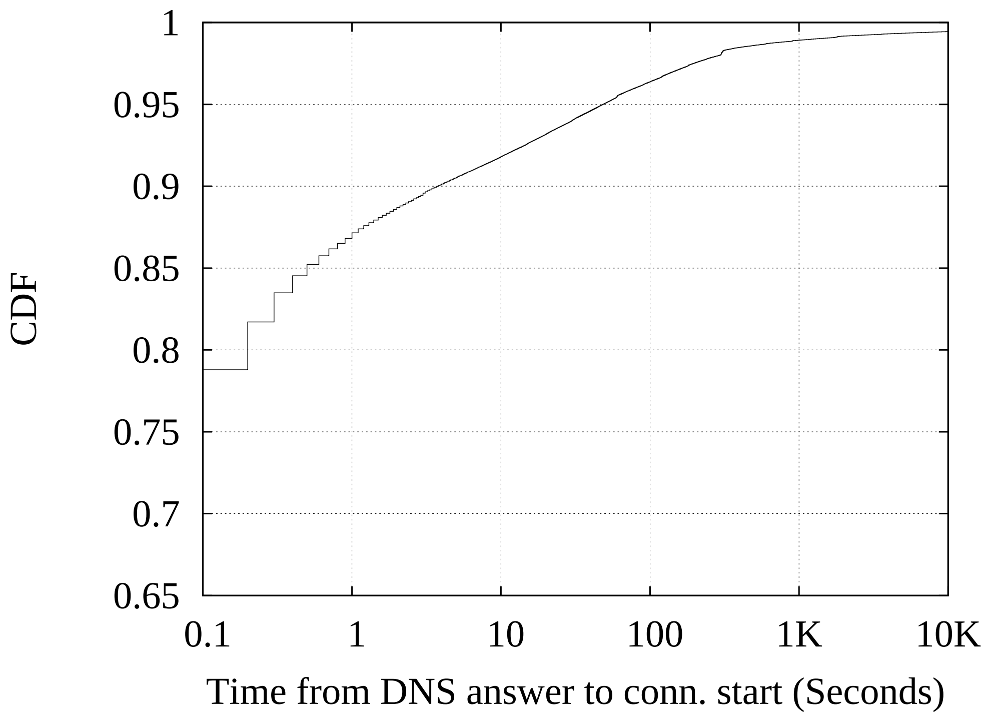


Figure 5.8: Time From DNS Response to First Connection

As discussed above, DNS responses often contain more than one IP address for a particular hostname. In principle, these IP addresses are equivalent and clients can use any one of the set. We now turn to understanding which IP addresses clients actually do use in subsequent communication. We first exclude all connections that follow from DNS responses that contain only one IP address. Then for each connection we determine the position of the destination IP address and the number of addresses in the corresponding DNS response. Figure 5.9 shows the percentage of use we find for each position within DNS responses, as a function of the number of addresses in the response. Each bar is divided into regions that represent the percentage of use for each position in the answer. For instance, the third bar represents all responses containing four addresses and is divided into four regions. The bottom region of each bar represents the use of the address in the first position with the address positions progressing up the bar with the top region representing the use of the last address found in the responses.

We find that more than 60% of the connections are made to the first IP address listed in the response across all response sizes with the exception of responses with five addresses—and, even in that case the first address is used in just under 60% of the subsequent connections. Further, for responses with 3–6 IP addresses the use of all but the first position is fairly uniform, which suggests that some applications are choosing randomly from the returned set of addresses. This uniformity generally holds for responses that contain 7–10 addresses, as well, with one exception. For responses with 8–10 addresses the exception is the last address listed—which is used more often than the others, suggesting a "use last" policy. For responses with 7 addresses the most common address (after the first) we find in subsequent traffic is the 4^{th} address. While we have no firm explanation for this behavior, we note that some common programming languages contain shortcuts to provide the first IP address in a DNS response [GNU, Ora].

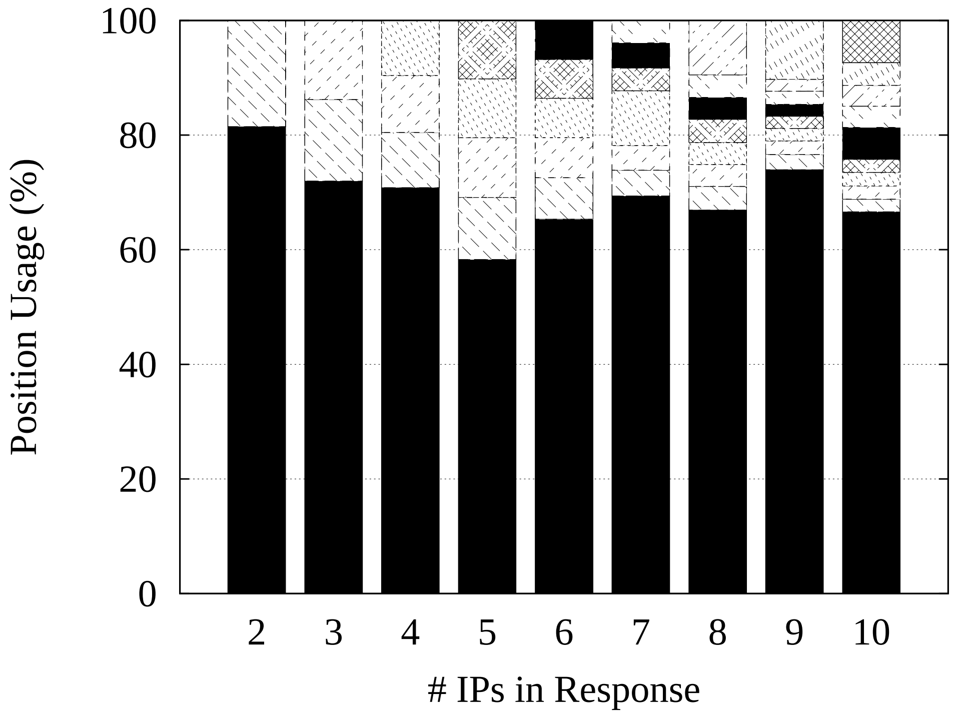


Figure 5.9: Used Answer Position

5.5.2 TTLs

We next focus on how well clients honor the TTLs given in DNS responses. Previous work observes that ≈47% of clients and their resolvers in combination violate the TTL and that most violations extend beyond two hours after the mapping expires [PAS$^+$04b]. Further, [SKAT12b] finds that 8–30% of connections utilize expired DNS records across several datasets from different vantage points. In our dataset we find that 13.7% of TCP connections use addresses from expired DNS responses—significantly less than indicated in [PAS$^+$04b]. We speculate some of these violations are due to modern browsers "pinning" IP addresses to hostnames in an effort to mitigate the impact of cross-site scripting attacks [JBB$^+$09]. Figure 5.10 shows the distribution of the elapsed time between a record's expiration and when it is used. Some connections come seconds after the expiration. These are likely unavoidable—having to do with an application looking up a short-lived mapping and then using it slightly after it expires or the expiration process of a local cache being somewhat coarse. We find, however, that roughly 60% of the violations come 1 minute—1 day after the expiration. The median elapsed time between expiration and use is 168 seconds. Again, our results show significantly smaller violation sizes than in [PAS$^+$04b], which reports that over 95% of violations exceed 1K seconds.

Despite our finding that TTL violations have decreased, a significant # of connections still violate the DNS TTL. We note that content providers and CDNs are often at odds with web browsers in this area—while content providers try to shape traffic at a fine granularity with short TTLs, web browsers violate these TTLs to increase security. Ignoring TTLs can work because even if a DNS mapping is no longer optimal for a particular client, the mapping generally continues to be valid well past the TTL expiration. Still, this points to a need for better security mechanisms that do not call for clients to violate protocols in order to protect themselves.

Finally, we revisit the length of TTLs in the context of the traffic that is derived from DNS responses. In § 5.3.1 we consider the TTLs set by content providers as a function of

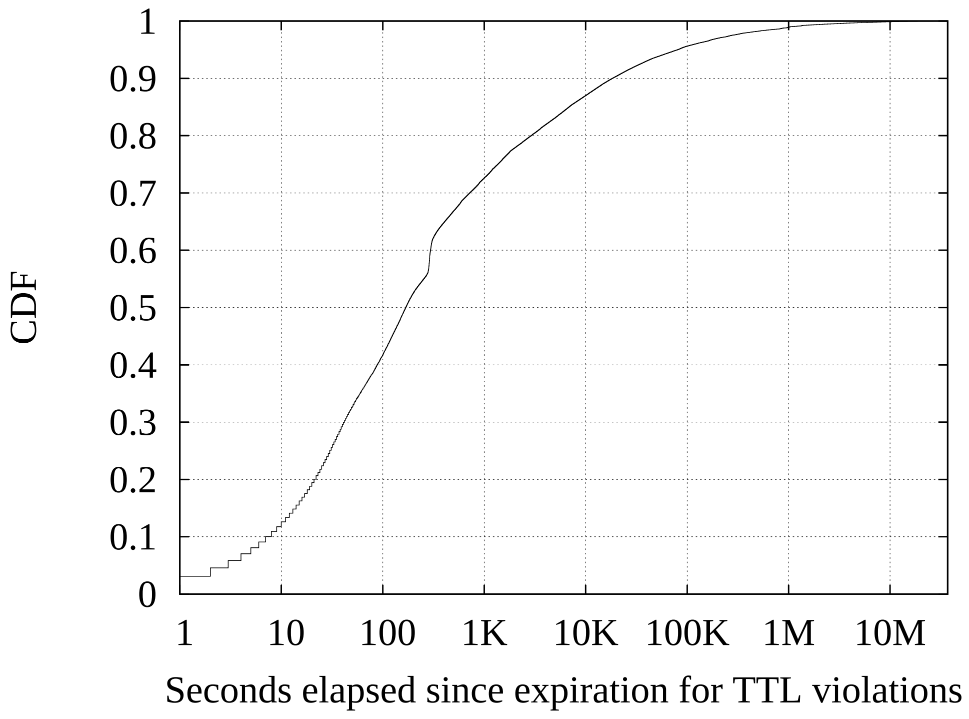


Figure 5.10: Distribution of TTL Violations

the hostname (Figure 5.2)—hence weighing all hostnames equally. We now take a different approach to understand TTLs based on client usage. For each maximum TTL we find in our dataset, we count both the number of connections made and bytes downloaded using the given TTL. This indicates the amount of traffic "shapable" at a given time granularity. Figure 5.11 shows the distributions of these two metrics as a function of maximum TTLs. We find that using these metrics skews the tail of the TTL distribution to lower values. While roughly 55% of hostnames have TTLs of at most 350 seconds, we find that 68% of connections and 87% of traffic volume is associated with DNS records with TTLs of at most 350 seconds. Further, while 80% of hostnames have TTLs of at most one hour (§ 5.3.1), we find that 85% of connections and 95% of bytes associate with DNS TTLs of at most one hour.

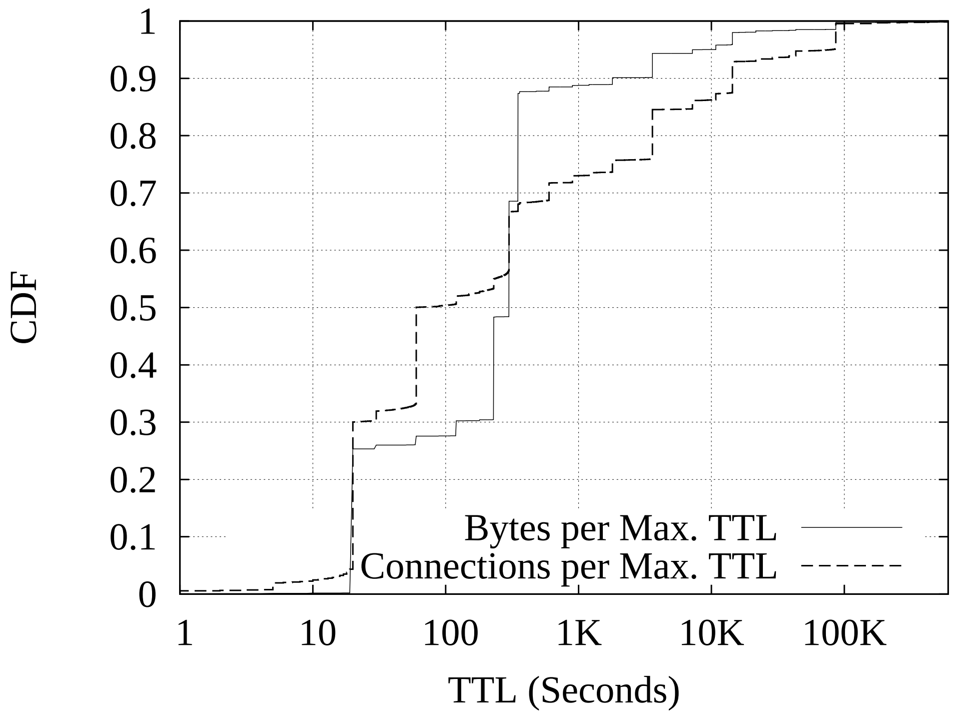


Figure 5.11: Weighted TTLs

5.6 Summary and Future Work

The contribution we make in this chapter is an empirical understanding of a broad range of characteristics of modern DNS behavior. Specifically, we focus on facets of DNS behavior that pertain to the phenomenon of enabling traffic engineering using DNS mappings. While previous studies have tackled empirical assessment of DNS, these studies were conducted many years ago. We believe that since the DNS is an evolving system, our reappraisal is timely. Furthermore, we note that our analysis covers aspects rarely or never seen during previous studies (e.g., prefetching).

While this work represents an updating of our understanding, we also stress that it is only an initial step. Our data comes from a small network and broader data would be useful. We have gathered data from larger networks, however, it is not from the same user-centric vantage point as we have in the CCZ—making it more difficult to understand client behavior from this broader data. That said, we are actively analyzing this additional data to gain understanding where possible. Finally, as discussed in § 5.3.4 we are starting to work on a replica selection model that we can apply to all traffic, rather than relying on manual case studies as we have herein.

Chapter 6

Communicating without Fixed Infrastructure

6.1 Introduction

The Internet has increasingly moved from a system used to disseminate information to users from a relatively small number of content providers to a system that facilitates sharing information among users. This style can be plainly found in the most popular destinations and applications: Twitter, Facebook, Flickr, Skype, BitTorrent, one-click file sharing systems (e.g., RapidShare), etc. The shift from merely consuming information to sharing information has in fact led to several efforts to change the basic model of networking from host-based to content-based [KCC$^+$07, JST$^+$09] as this latter has become the basic mode of operations for users. That is, users fundamentally do not want to access some host in the network, but rather want to swap a given piece of information. The techniques explored in this work strive to transfer small amounts of information using a scheme that is not fundamentally host-centric.

In a network model where information is generally disseminated upon request, we can readily build highly robust systems. A user interested in buying a book can easily find a book seller using a well-known DNS name (e.g., “amazon.com”). Further, server farms, content delivery networks, replicated DNS servers, geographically disparate replicas, multi-homed connectivity, etc. provide robustness of operation. We refer to this as the *central hub* model. Even if physically distributed, the service is orchestrated at some logically central location. This model makes perfect sense for certain activities (e.g., legitimate e-commerce).

However, as noted above, users have evolved to become the most prolific content providers on the Internet. In technological terms this shift has manifested in one of two basic ways: (i) using a central hub to connect users and hold the shared content (e.g., Twitter) or (ii) using a central hub as a bootstrapping mechanism for direct peer-to-peer information exchange (e.g., a BitTorrent tracker or, in the tracker-less variant, a site listing an existing DHT node). While the role of the central hub is reduced in the second approach, it is still required. Although a lightweight central hub may be perfectly reasonable in some cases, there may be other cases where such a central presence is undesirable, such as:

- For peer-to-peer systems, requiring a central hub to bootstrap establishes a system vulnerability that can hamper operation even though the major functionality is distributed at the peers. For instance, if the central hub loses connectivity (power, etc.) the larger system would likely be still functional if not for the inability to bootstrap. Therefore, for robustness reasons, not depending on a fixed central hub is useful.
- Another aspect of using a central hub is that it provides a tangible choke point that can be readily blocked by policy. For instance, blocking a large BitTorrent tracker could affect many peers even though the peers themselves do most of the work to exchange files independently from the tracker once bootstrapped. Another example is the recent case of Egypt disconnecting its major ISPs from the broader Internet—which effectively disconnects users from myriad central hubs. However, if local connectivity remains, users could in principle bootstrap to communicate locally even though their usual means for doing so is disrupted.

This situation begs a question: *Can we increase robustness and flexibility of information sharing services by allowing consensual actors to initiate communications over the network without a central hub?*

Within a small area this is straightforward. For instance, within a broadcast domain one party could encrypt a message with a secret that was pre-arranged with the recipient(s) and then broadcast the message. The intended recipient(s) would be the only ones who could make sense out of the message. Further, there is no direct targeting of the recipient(s). While such a scheme is trivially possible it does not address our question when we scale beyond individual broadcast domains. However, this limited scheme provides for a model of sorts for solving the problem in a broader way.

In this work we develop a global *covert broadcast domain* that allows actors with only a simple shared secret to exchange small messages without the secret ever being directly used within the network (and thus itself becoming a central hub, of sorts). This message could be self-contained information or a way to bootstrap further communication. We develop this covert broadcast domain by using standard DNS servers to hold information not in the traditional sense of serving records, but by leveraging the caching behavior of the servers to convey information. Further, the scheme is not dependent on any particular DNS server, but rather any DNS server the actors agree on (as discussed in § 6.2). In other words, we design a technique that factors out the need for a well-defined central hub for information sharing and/or bootstrapping.

We explain the mechanism in detail in subsequent sections. However, as a touchstone the reader can think of breaking a message into its component bits. Each bit is represented by a cached record in an arbitrary DNS server the actors have agreed upon. The value of each bit is represented by the returned TTL value of the DNS record—e.g., the one bits may have a TTL 10 seconds larger than the zero bits. In this way we use DNS servers' natural capabilities of caching and aging records to encode ephemeral information in the system without relying on any particular fixed infrastructure or name. In the remainder of this chapter we show we can accurately publish and query for such information.

6.2 DNS Server Discovery

As sketched above, we leverage caching DNS servers to hold information. To fulfill our vision, the first premise is that there should be many DNS servers on the Internet that will hold the messages we seek to store. We note that in Chapter 4 we found over 1 million open DNS servers that might fulfill this requirement. Further, actors should be able to independently discover common DNS servers to hold the exchanged messages. Therefore, before we embark on storing and retrieving information from DNS servers we perform a DNS scanning experiment to understand the prevalence of usable DNS servers.

Actors wishing to exchange messages must share a secret S. This is used in a number of the tasks in our overall procedure, and in particular for finding common DNS servers. While we consistently refer to S as a "secret", we note that nothing compels the communicating actors to keep S strictly private. Rather, S must be shared and S is only needed by the endpoints of the communication and not the DNS servers. For example, a BitTorrent application could maintain a hard-coded collection of shared secrets for client use. From S we define a generator function as:

$$G(x) = sha1(sha1(S) + x), \tag{6.1}$$

where "+" denotes the append operation and x is a string. By running $G(''IP1'')$ and using the low-order 32 bits as an IP address any actor holding S can independently derive the same series of addresses—by replacing "1" in the call to $G()$ with linearly increasing integers—to find a usable DNS servers to mediate their communication. Each host in the common list is probed with a DNS request for a name within a domain that we know to enable wildcards[1]. With DNS wildcarding, a zone administrator defines a default mapping

[1]In particular the name we use is "dns.research.project.visit.dns-scan.icir.org.if.problematic.HASH.ws"

to be returned when one has not been explicitly provided. For instance, the administrator of "case.edu" might define a wildcarded hostname of "*.people.case.edu" with an IP address of "1.2.3.4". In this case, any query ending in ".people.case.edu" will return "1.2.3.4", unless the specific hostname queried is also defined.

In our scanning experiment we probed from roughly 80 PlanetLab [CCR$^+$03] nodes[2]—each using its own random S—at a rate of 2 DNS queries/second. A correct response from a given host is used to trigger two more queries of the same server to ensure the TTL is being decreased on subsequent retrievals.[3] Assuming the TTL is being correctly decremented, we consider the server to be usable. However, as we discuss in subsequent sections, it is not unusual for a DNS server to pass this initial set of checks only to display non-conforming TTL behavior during message publication.

Across 22.7 million probes we find that the hit-rate is approximately 0.4%. The median number of probes sent between identifying subsequent servers is 194, while the mean is 281. Further, we find the maximum probes sent before identifying a server is nearly 9,000, with the 99th percentile being 1,284. These results suggest probing is tractable for our purposes because even scanning at a low rate will identify multiple servers. E.g., sending one probe per minute over the course of one day will yield five DNS servers on average. DNS servers that disappear after discovery (as has been noted in Chapter 4 and elsewhere in the literature [DPLL08b, LL10b]) will be readily detected as attempts are made to store information at the given server. Such knowledge can be used to trigger a new server detection phase.

While for this work we use relatively low rate scans for all our experiments—at most 10 queries/second—we note that using a higher scanning rate could be possible in some circumstances and would allow us to find usable DNS servers more rapidly. For

to be up-front about what we are doing should our experimental queries trip alarms.

[2]PlanetLab often experiences node churn and, while we tried to choose reliable nodes, the number of nodes used in each individual test throughout this chapter varies slightly.

[3]We found in Chapter 4 that some DNS servers do not decrement the TTL of their cached records, leading to this test.

instance, in a small experiment we were able to identify 60 recursive DNS servers within 15 seconds using a residential cable modem connection.

An alternative to random scanning is hit list scanning (as covered in a general way in [SPW02]). Actors could agree on some independent list of servers to scan. For instance, Alexa.com tracks web site popularity and the authoritative DNS servers connected with the listed domain names could be checked for suitability for our purposes. We probed the authoritative DNS servers associated with Alexa's top 10K web sites to determine if they would respond to arbitrary recursive queries from outside hosts and found a hit rate of approximately 3.3%—or an order of magnitude more than in our random scanning experiments. We note that each server requires an additional probe to determine the IP address of the authoritative DNS server that corresponds to a given name when compared with the random scanning approach given above. While the success rate means that the hit list mechanisms represent a healthy reduction in the number of probes, we cannot say whether the reduction is enough to fly under the radar.

Finally, we note that such a hit list approach runs counter to the notion developed in § 6.1 of not requiring a central hub to bootstrap communication since actors must be able to access the hit list. However, this approach can first be viewed as an optimization and is not strictly necessary. In addition, we believe a variety of hit lists can be used—e.g., from addresses in mailing list archives, using web sites found in the Twitter public timeline over the course of some time period, etc. This makes correlating the DNS probing activity with the hit list more difficult. The DNS requests sent via a hit list are also more likely to look legitimate on an individual probe basis because they are actually connecting with DNS servers as opposed to most of the probes in the random scanning experiment which do not hit active servers. Also, since the hit lists come from uninvolved actors they may be more difficult to block without shutting down some useful functionality; however, this varies with the general popularity of the source of the hit list (e.g., blocking Alexa may cause relatively little harm, but blocking Twitter may cause an unacceptable loss of functionality).

6.3 A Basic Bit Channel

As described above, our goal is to utilize DNS servers' natural ability to cache and age information to store small messages without directly inserting records into the DNS system. As an initial use case, we consider publishing a 32-bit IPv4 address using this system as a basic bit channel. We start with this use case because a bit pipe is the most basic communication channel. Further, we presume that once known, an IP address can be used to form the basis of higher layer communication. In this section we use the procedure outlined above in § 6.2 to find suitable recursive DNS servers and, as they are found, publish and retrieve 32-bit messages.

6.3.1 Procedure

The process of storing messages in DNS servers starts with a pre-arranged secret S between all parties involved in the communication. Using this secret, we define a generator as shown in equation 6.1. We also need a domain we know to support wildcard DNS queries, that is, queries for arbitrary names within the domain still return some record. As will become clear, we also need the domain to assign a sufficiently large TTL to its DNS responses. Domains supporting wildcards are widespread [KGPP10], and we found that many also return TTLs sufficient for our needs. In all our experiments we use the ".ws" domain (an arbitrary choice that returns TTLs of 3 hours). Note, we consider alternate designs that do not have this requirement in § 6.6.

Let M be the message we wish to transmit and M_i be its i^{th} bit. We now outline two procedures for encoding M within a DNS server D.

TTL Method

The first method we employ is based on inserting records corresponding to all bits in M in such a way that the zeros and ones are distinguishable by the TTLs returned in lookup responses after publication. The publication process proceeds as follows:

1. We generate a name for each bit M_i of the message using:

$$R_i = G(\text{"}Record\%d\text{"}, i), \tag{6.2}$$

   where “Record” is just an arbitrary identifier that all actors involved know (here and in the rest of the chapter we use a `printf`-like notation to compose strings).

2. Similarly, we generate a “barrier record” using:

$$B = G(\text{“}Barrier''), \tag{6.3}$$

3. Next we form sets, Z and U, where i is inserted into Z if $M_i = 0$ and U otherwise.
4. For each $j \in U$ we execute a DNS request to D for the hostname “R_j.ws”, retrying until a response is received for each record.
5. We next pause for roughly five seconds. The choice of five seconds is arbitrary. The value needs to be more than one second as that is the granularity of DNS’ TTL. We leave optimizing the publication time as future work.
6. We then execute a DNS request for “B.ws”, retrying if necessary.
7. We again pause for roughly five seconds.
8. For each $k \in Z$ we execute a DNS request to D for the hostname “R_k.ws”, retrying until a response is received for each record.

The general idea behind this process is that D will cache the requested records with associated TTLs that originate from the authoritative server. Given the publication pattern all R_i records that have a TTL shorter than that of the barrier record B correspond to $M_i = 1$ and records having TTLs longer than that of B correspond to $M_i = 0$.

The data retrieval process borrows steps 1 and 2 from the procedure outlined above. We then query for "B.ws" and each record in R, recording the TTLs for each returned record as B^T and R_i^T. We then set M_i' to one if $R_i^T < B^T$ and zero otherwise. At this point the retrieved M' should be equivalent to the message M that was published.

Recursion-Desired Method

The second method was sketched in a Black Hat presentation [Kam04]. To our knowledge, there has never been any experimentation to determine whether the scheme works or how effective it may be. The procedure—which we denote the "RD method"—works as follows:

1. We generate a name for each bit M_i of the message using:

$$R_i = G("Record\%d", i), \tag{6.4}$$

2. Next we form a set U, where i is inserted into U if $M_i = 1$.

3. For each $j \in U$ we execute a recursive DNS request to D for the hostname "R_j.ws", retrying until a response is received for each record.

The general idea behind this approach is that the records corresponding to the one bits in M are cached by D, whereas the zero bits are not encoded in D in any way. We leverage this when retrieving the data by sending queries for each of the 32 names in R with DNS' "recursion desired" flag set to false. This indicates that D should only look in its own cache for the given name and not recurse up the DNS hierarchy to resolve the given name. We initialize an M' to all zeros and then any "R_j.ws" query that returns a valid response indicates that M_j' should be set to one. After considering each of the 32 records M' should be equivalent to the original M.

	TTL Method	RD Method
Pub. Attempts	125K	87K
Unusable Servers	21K	15K
Non-Responsive Servers	742	520
Non-Recursive Servers	2.6K	1.7K
Non-Decrementing TTL	15K	8.9K
Weird TTL Decrementing	2.8K	898
Ignores RD=0	N/A	2.6K
Usable Servers	104K	72K
Successful	92K	58.8K
Failure: Packet Loss	3.6K	4.8K
Failure: No Data Found	3.6K	3.3K
Failure: Corrupt Data	5.0K	5.4K

Table 6.1: Bit-Pipe publication results

6.3.2 Results

We test the accuracy of both publication mechanisms described above by storing a 32-bit message (a la an IPv4 address for bootstrapping) in a DNS server and then attempting to retrieve the message. We use the procedure outlined in § 6.2 to probe for DNS servers and upon finding each such server we publish a message and then attempt to retrieve it. Each publication strategy is tested in its own scanning pass (which run in sequence, not parallel). We use roughly 80 PlanetLab nodes for the test. Each node performs independent scans to identify DNS servers and start the subsequent tests.

After each publication the host waits 10 seconds and then retrieves the message to assess the efficacy of the data insertion process. Table 6.1 shows the results of our publication attempts. First we note that in spite of our efforts (sketched in § 6.2) to identify unusable DNS servers during the scanning phase we still end up with problems in roughly 15% of the servers. The largest problems come from TTL decrementing issues[4]. We note that when using the RD method, we find servers that ignore the RD=0 setting in our requests. This would be trivial to also exclude in the scanning phase and in future efforts based on the RD method we would certainly do so. While these problems do not speak to

[4]Non-conforming TTL handling does not render a server unusable in the case of RD method; however, we wanted to compare both methods over a similarly-selected set of DNS servers.

the efficacy of our information sharing technique, they do illustrate that one must exercise care in choosing suitable DNS servers.

The lower portion of the table shows the results for usable servers. We find a publication success rate of roughly 88% for the TTL method and 81% for the RD method. The largest and most problematic publication issue is data corruption. Whereas the other issues listed in the table—no data found and packet loss—can be readily identified during the retrieval process, corrupted data gives no outward signs of problems. As we have designed a generic bit pipe, it would be possible to apply Forward Error Corruption (FEC), or simply a parity bit, to the bit-stream to reduce the number of corruption errors (at the expense of requiring more bits, of course). Also, we note that packet loss is an issue—even though we re-try pending queries every two seconds until we receive a response or have transmitted four queries. The final problem of no data being found likely comes from DNS servers that recursively lookup records, but do not cache names (for long).

We next turn to a more general investigation of information retrieval. For the successfully published messages via the TTL method we scheduled retrievals from a set of 55 different PlanetLab nodes chosen via round robin across our list of roughly 80 PlanetLab nodes. We schedule five retrievals (from different nodes) at each of eleven intervals between 10 seconds and 128 minutes after publication. This methodology allows us to test both (i) whether the information is available to a breadth of hosts around the network and (ii) the storage longevity we can expect from the mechanisms.

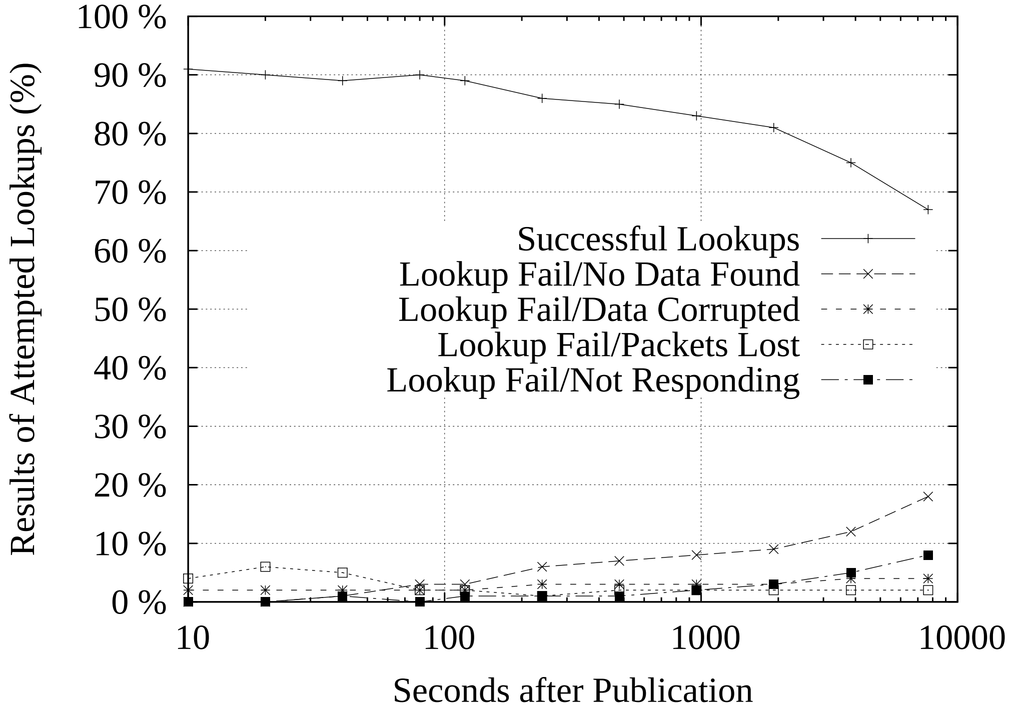
For the RD method we change to using a single, randomly selected PlanetLab node other than the publishing node (as opposed to 5) for each lookup interval. We analyze our TTL method lookups and found that in 89% of cases, all nodes return the same result. In a further 7% of cases, the only disagreement was due to packet loss—e.g., 4 nodes successfully received the message, while 1 node failed due to packet loss. Only 4% of all TTL method lookups exhibit two or more nodes reading different values from the server. As we found few disagreements among nodes, using only a single node for retrieval enabled us to reduce network traffic and simplify the experiment.

Figure 6.1 shows the results of retrieving the information we published as a function of time since publication. We note that just after publication the TTL method shows a roughly 90% retrieval success rate, whereas the Recursion Desired method is nearly flawless. The success rate two hours after publication drops to roughly 70% for both methods. As shown in the plot, the predominant cause of the dropoff in success is an increase in the instances of not finding the data on the server as the time since publication increases. This is a natural result of names being evicted from the cache. Even though the names nominally have TTL left, the names are to a large extent not being used and so it is natural that some LRU-like policy would evict the names corresponding to all our queries. We also note that failures to contact the DNS server rise with the time since publication, likely due to the transient nature of many of these DNS servers (as shown in Chapter 4). The remainder of the failure causes remain fairly constant and relatively small across the time period.

Finally, to ensure that the PlanetLab platform itself was not biasing our results in some fashion we replicated the above experiments from a host at ICSI. While the PlanetLab retrievals were conducted from 55 disparate machines we used the same machine at ICSI for all aspects of our tests (scanning, publication and retrieval). The results from the ICSI runs are consistent with the PlanetLab experiments. The retrieval results are similar with the RD method showing higher success soon after publication than the TTL method, but both dropping off over time. The predominant cause of failures over time is finding no data on the DNS servers, just as we find in the PlanetLab results. Therefore, overall we conclude that the PlanetLab platform itself is not significantly biasing the conclusions we draw from our experiments.

6.3.3 Discussion

We now briefly touch on additional ways to enhance the basic bit pipe we have constructed.



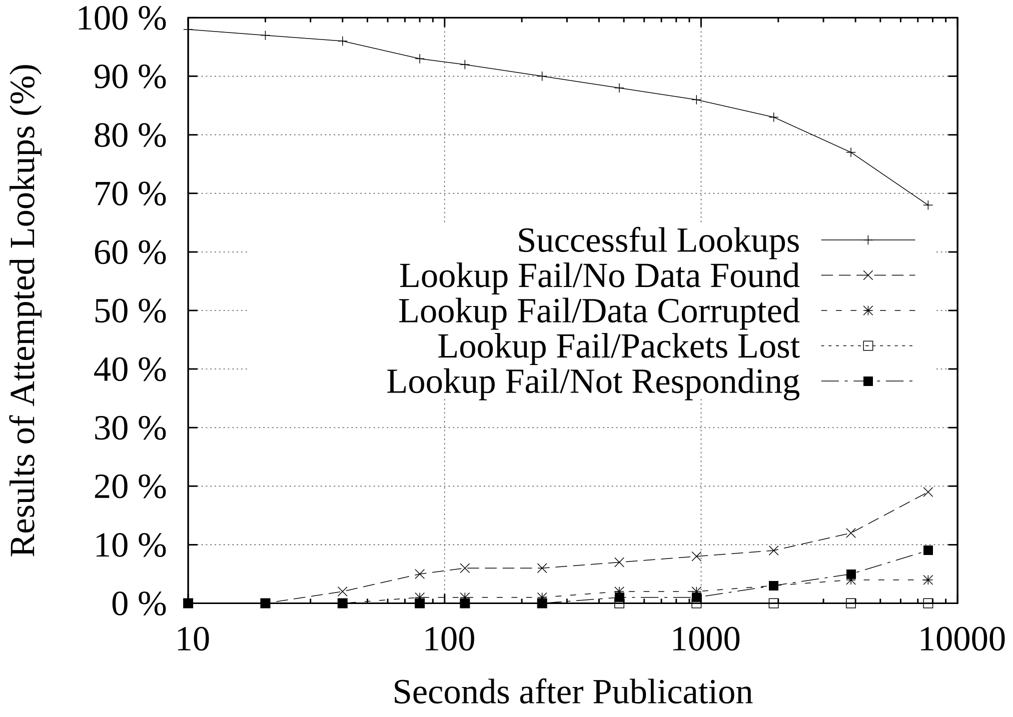


Figure 6.1: Bit pipe retrieval results for the TTL (top) and Recursion Desired methods.

Robustness

A traditional way to make a bit channel more robust is to add coding to the message. For instance, a Reed-Solomon code that doubled the size of the message (to 64-bits) could detect any bit error and correct up to 16 bit errors in the message. For corrupted retrievals, we find that not more than half of the message is corrupted in 8.7% of cases for the TTL method and in 2.3% of cases for the RD method. Coding would also help reduce the impact of losses in our results.

Widening

A natural way to widen the channel in the TTL method is to add more barrier records, which allows for more symbols to be transmitted. For instance, using two barrier records we could encode three symbols—enough to encode messages in Morse Code (using dots, dashes and spaces). We explore this further in § 6.4. The RD method is not directly amenable to widening due to the reliance on a fundamentally binary property of the system (namely the RD flag).

Synchronization

Note that messages in the system have a higher probability of being successfully retrieved within the first several minutes after publication. While retrievals further out in time have reasonable success rates, it may behoove some uses of such a channel to roughly synchronize publication and retrieval. For instance, when swapping S out-of-band, actors may also agree that publications will take place at the top of every hour. Even with imperfectly synchronized clocks this could increase the chances of successful message transmission.

Collisions

One issue with a single secret is that if multiple actors are publishing to that secret they will corrupt each other's messages. A straightforward way to deal with this is to assign roles to actors with respect to a particular secret during the secret exchange. For instance, for some secret S_1 Alice may be designated as the publisher and Bob the recipient, while the opposite could hold for a second secret S_2.

6.4 Widening The Channel: A Twitter-like Service

Our focus in the last section was a basic bit pipe that can be constructed through DNS for rendezvous purposes (as discussed in § 6.1). In this section we tackle the problem of widening the channel, asking: can we widen the channel enough to encode actual message contents in the DNS? This would allow for message exchange without dedicated infrastructure or central hub, which—as discussed in § 6.1—is sometimes useful to avoid policy constraints or because the system's entire infrastructure is not reachable. To answer our question we design a Twitter- or SMS-like service that can convey 140 character messages using recursive DNS servers. We use two different mechanisms for conveying these messages. The first technique is a simple extension of the barrier record mechanism used in the last section that accommodates more than two symbols. The second mechanism uses the DNS TTL values on various records as implicit barrier records.

We start both schemes with a dictionary of 56 symbols (enough for letters, numbers and several additional punctuation marks, etc.). Each symbol is mapped to a value (1–56) with the mapping known by all actors. As in § 6.3 we rely on a secret S, a generator function $G(x)$ (defined in equation 6.1) and a recursive DNS server D. The 140-character message we wish to send is M, with M_i denoting the i^{th} character in the message (all characters are assumed to have been mapped into our custom 56 character dictionary).

6.4.1 Procedure

We now outline our explicit and implicit methods for storing content within DNS servers.

Explicit Barriers

Our first channel-widening procedure simply adds more barrier records to the TTL method discussed in the last section to differentiate more symbols. The procedure follows several steps:

1. For each possible data value in our dictionary (1–56), we generate a corresponding barrier record:
$$B_i = G(\text{"}TwitterBarrier\%d\text{"}, i) \tag{6.5}$$
2. We also generate names for each character of the message (1–140) as above:
$$R_j = G(\text{"}TwitterQuery\%d\text{"}, j) \tag{6.6}$$
3. Finally, we iterate through the symbol values from $k = 1 \dots 56$. For each k we find the set C of all characters in the message whereby $M_x = k$. Then for each index $x \in C$ we query D for the hostname "R_x.ws"—ensuring we receive a response—which inserts all characters with symbol k into the DNS. We then wait two seconds and query for B_k to insert the barrier between symbol k and symbol $k+1$. We then wait an additional three seconds before repeating this step for $k+1$.

Data retrieval borrows steps 1 and 2 from the publication process above. We then query for all B_i and R_j records and store the corresponding TTLs for each as B_i^T and R_j^T. For each R_j^T we find the k such that R_j^T falls between B_k^T and B_{k+1}^T. We then assign the j^{th} element of the new message M' as $M'_j = k + 1$.

One downside to this approach is that the publication time requires 5 seconds per symbol in our dictionary (or 280 seconds for our 56 symbol dictionary). We may be able to decrease this time, but the process fundamentally depends on a noticeable interval between queries and given DNS' TTL is measured in units of one second the publication process will still remain lengthy for any reasonable dictionary size. Another downside is the requisite extra barrier records that convey no content directly. This means that in our case of 56 symbols and a message length of 140 characters, only approximately 70% of the records contain content.[5]

Implicit Barriers

Above we rely on a record's TTL relative to explicit barrier records to determine the value of the symbol. An alternative is use the difference between the TTLs of individual character records and some base record to provide the value of the given character. For instance, if we wanted to encode the 15^{th} symbol in our dictionary we would publish the record 15 seconds after some base record such that the TTL of the record upon retrieval is 15 seconds greater than the base record. While such a procedure is intuitive it is also brittle in that the timing has to be quite precise. In actuality we use the following procedure to publish a message:

1. We make one name for each character of the message using character's position i using:
$$R_i = G(''TwitterQuery\%d'', i) \tag{6.7}$$
2. Similarly, we generate a "base record" using:
$$B = G(''TwitterBase'') \tag{6.8}$$
3. For each M_i we set $M_i = M_i \times 4$. This spreads the values into 4 second windows such that we build robustness to timing and TTL decay issues. I.e., anything falling into a given 4 second time window will be treated as the given character.
4. We send three back-to-back queries for B to DNS server D in a (in an attempt to

[5]Note, an optimization to decrease the required publication time would be to order our symbol dictionary by character popularity such that unpopular characters would be concentrated at the end of the process. In this manner, the entire publication process might not have to be completed.

ensure one arrives in a timely fashion) and record the time of the first transmission as t. This signifies the beginning of the publication process and will be used as the basis for the content records to be inserted.

5. Each of the 4 second windows represents a particular value in our dictionary and all characters with that value are inserted within the window (with re-retries as necessary and allowed by the window length). Therefore, for each M_i, we schedule a DNS request to D for the hostname "R_i.ws" at time $t + M_i + 1$, $t + M_i + 2$ and $t + M_i + 3$. The latter two are retries, which are canceled if the previous request returns a correct response prior to their execution.

Since we have 56 symbols in our dictionary and leverage 4 second windows the publication process takes 224 seconds. As in the explicit case, we may be able to get additional savings by decreasing the window length. However, it is fundamental to have separation between the symbol values. The implicit scheme does offer a savings in terms of the number of non-data records required compared with the explicit version above. The implicit scheme requires only one such record, B.

The data retrieval process borrows steps 1 and 2 from the publication process above. We then query for B and each record in R, recording the TTLs for each returned record as B^T and R_i^T. We calculate the value for each record as $(R_i^T - B^T) \div 4$ and store that in M_i'. After M' is formed from the 140 component characters it should be equivalent to the original M if the procedure worked as intended.

6.4.2 Results

We coupled our Twitter-like message exchanges with the DNS scanning outlined in § 6.2. Once we identify a recursive DNS server via scanning we attempt to publish a message to that server. The explicit and implicit mechanisms were tested independently. I.e., we run a scan coupled with the explicit method followed by another scan coupled with the implicit

	Explicit	**Implicit**
Pub. Attempts	80K	86K
Unusable Servers	10K	13K
Non-Responsive Servers	461	298
Non-Recursive Servers	981	5
Non-Decrementing TTL	8.5K	11K
Weird TTL Decrementing	816	1.4K
Usable Servers	70K	73K
Successful	53K	47K
Failure: Packet Loss	6.6K	6.3K
Failure: No Data Found	1	1
Failure: Corrupt Data	10K	20K

Table 6.2: Twitter-like publication results

method. As a first test after publishing our messages we re-request all records pertaining to that message spaced over 12 seconds (such that we do not overload the server) to judge the success of the publication operation.

Table 6.2 shows the results from our publication of 140 character Twitter-like messages. As was the case in our bit-pipe experiment, the table shows that despite our usability tests during scanning, we still find a sizeable number of servers that exhibit problems when trying to leverage them for publishing short messages. The middle section of the table shows that around 14% of servers are unusable in both experiments. The bottom section shows the results from the usable DNS servers. We find the publication success rate to be 76% and 64% for the explicit and implicit mechanisms, respectively. The difference in success rates is caused nearly entirely by data corruption with the implicit barriers showing twice the number of failures as the explicit barriers. This illustrates that TTL decrementing procedures on DNS servers are often gross enough to cause ambiguities in our 4 second time windows. While this also happens with characters running into barrier records in the explicit case the reliance on relative timing between explicit records is found to be more robust—at a cost of 30% more records in the system, as discussed above.

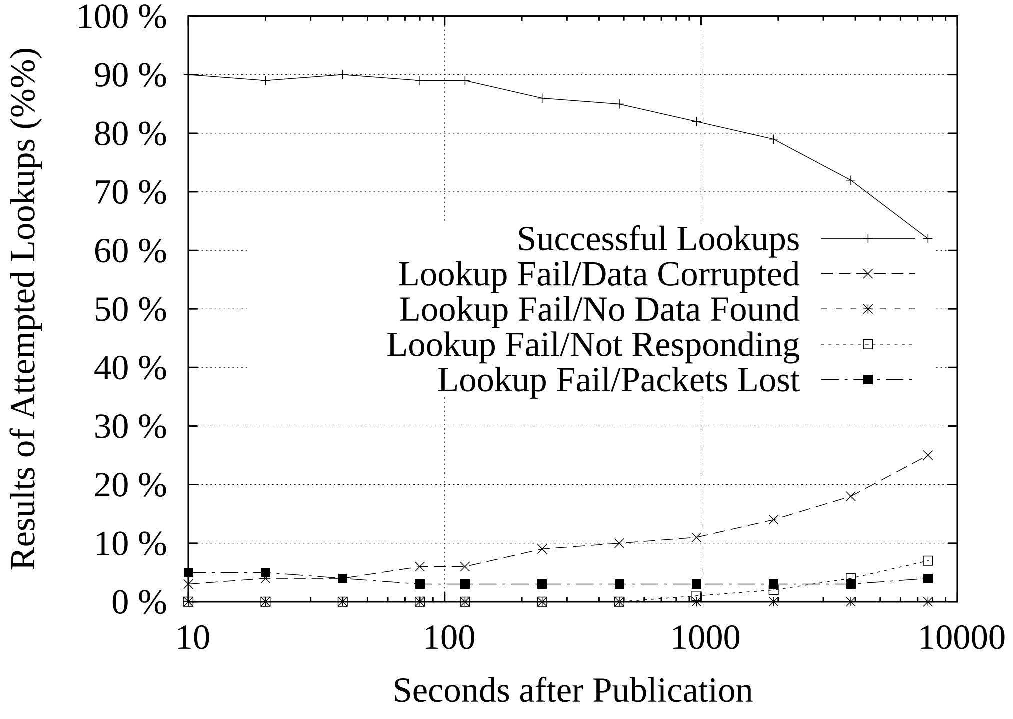
As explored in more detail below, natural language messages like Twitter or SMS messages can often retain their overall meaning in the absence of small amounts of loss and/or corruption. In the investigation of message retrieval below we consider only perfectly published records (i.e., the successes listed in table 6.2). However, this could be relaxed in a real system. For the messages that experienced loss we find the median number of losses to be 3 and the mean to be roughly 15 across both methods. Arguably this often leaves us with a usable message. However, when we find a message to have been corrupted we find the Levenshtein edit distance [Lev66] between the original message and the stored message to be over 90 at the mean and median across methods. Such a mangled message is obviously unusable.

For the publication attempts classified as successful above, we engage with 55 additional PlanetLab nodes to retrieve the messages at various times from various places just as in § 6.3. The results for the explicit and implicit methods are shown in Figure 6.2.[6] For both mechanisms we see similar behavior. The success rate of retrieving the published information within several minutes of publication is around 90%. The last data point shown is over 2 hours after publication and shows the retrieval success rate has fallen to under 65% in both cases—with the explicit mechanism slightly lower than the implicit scheme.

Data corruption—i.e., receiving a wrong character—is the predominant reason for failure in both methods—especially in the longer intervals. Since the corruption rate increases for both schemes over time, this indicates that at least some servers clearly evolve the TTL of cached records in a way that corrupts the message. As previously noted, an advantage of sending Twitter- or SMS-like messages is that small bits of corruption can generally be absorbed without the message losing its overall meaning.[7] We therefore look at the retrieved messages and determine that if the message experienced some corruption but the Levenshtein edit distance [Lev66] between the original message and the retrieved message is less than ten (roughly 7% of the message) the overall message likely retains its usefulness and so term these retrievals "nearly successful". We find that the percentage

[6]Note, due to a measurement glitch the first two time intervals for the implicit mechanism are not available. We expect these results will be highly similar to those reported for the 40 second interval.

[7]This is not universally true as the difference between "1am" and "9am" is the difference of one character, but much time, for instance.



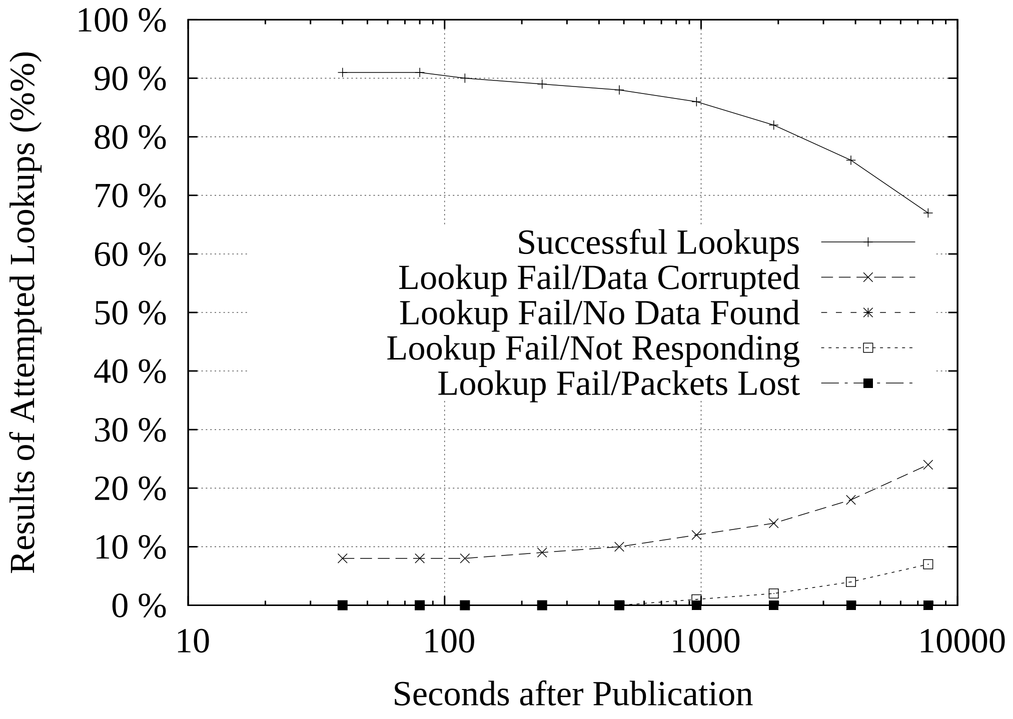


Figure 6.2: Explicit (top) and implicit retrieval results.

of messages that are nearly successful is roughly constant over time at 2% and 6% for the explicit and implicit schemes, respectively. One reason the explicit scheme may produce records that are more corrupt than the implicit version is that if one barrier record gets somehow perturbed (e.g., ejected from the cache), that will render all records the original barrier was meant to identify as corrupt. Therefore, such errors have a magnifying impact.

While both mechanisms we test show an increase in loss over time, the explicit mechanism falls prey to more messages experiencing loss in the longer time intervals compared with the implicit version. We attribute this to the additional queries that are required by the explicit mechanism making it more likely to experience a loss. As with corruption, we consider a message to be "nearly successful" if the loss is at most about 7% of the message. Similar to the corruption results, we find that the percentage of messages that are nearly successful is roughly constant over all time intervals tested. We observe roughly 1.5% and 0.25% nearly successful rates for the explicit and implicit mechanisms, respectively.

Adding "nearly successful" messages due to corruption and loss, we observe that for the first several minutes after publication the successful retrievals (shown in Figure 6.2) and the nearly successful retrievals hover around 95%—slightly less for the explicit and slightly more for the implicit methods. This is an approximation due to our data collection which did not record corruption information for messages containing loss. Therefore, it is possible that the messages classified as nearly successful due to a small loss rate could turn into failures when also adding corruption information into the judgment. Therefore, the percentage of nearly successful messages that experienced loss given above (1.5% and 0.25% for the explicit and implicit methods, respectively) should be taken as an upper bound on the fraction that might be so considered when also taking into account corruption information.

As described in § 6.3 we conducted a set of experiments from ICSI that did not involve PlanetLab in an attempt to ensure the PlanetLab platform itself was not biasing our results. For the implicit and explicit methods the experiments from ICSI show the same general trends as found in PlanetLab. In particular, roughly 90% of retrievals are successful shortly after publication with the success rate dropping over time. Additionally, the predominant source of unsuccessful retrievals is corruption which also aligns with our PlanetLab measurements. This double-check suggests the PlanetLab platform is not overtly impacting our results.

6.5 Eliminating Reliance on Wildcarded Domain Names

The DNS names we use in the preceding experiments are generated by the end hosts and not known to any third party. In Sections 6.3 and 6.4, we rely on wildcarded DNS domains to generate cacheable DNS records. In this section, we investigate an alternative possibility: using DNS names that do not exist. The DNS allows negative answers to be cached [And98a] by a RDNS just as a regular answer. Furthermore, [And98a] calls for negative DNS responses to include the "SOA" record for the zone issuing the response, and for this SOA record to be returned from the cache on each subsequent reply. We use the TTL value of this SOA record to recreate the "TTL Method" experiment performed in Section 6.3.1. We conduct a separate study on this "SOA TTL" method as we do not expect devices to cache negative responses as reliably as positive lookups. Furthermore, as discussed in Chapter 4, a large number of DNS servers overwrite NXDOMAIN responses with fictitious answers. Finally, we note that some software provides separate configuration options [Zyt] for the caching of positive and negative DNS answers.

6.5.1 Procedure

We first address our scanning methodology. We choose IP addresses to probe in the same manner as in Section 6.2. However, instead of sending a DNS query for a name that exists, we send a DNS query for a name we know not to exist. When we receive a response, we use slightly different criteria to evaluate server usability. In Section 6.2 we searched for IP addresses that would successfully resolve our query and properly decrement the TTL on subsequent queries. Here, our criteria for classifying an IP as usable include receiving the expected NXDOMAIN response code, finding an SOA record in the response, and observing the SOA TTL decrement on subsequent accesses. Across 33 million probes we find our hit rate to be approximately 0.26%—two-thirds the hit rate observed in Section 6.2. We find the median number of probes sent between identifying usable servers to be 249 with a mean of 364. The 99^{th} percentile number of probes sent before identifying a server is 1672, compared with 1,284 in Section 6.2. While scanning for servers compliant with the SOA method is more difficult, we do not believe the increased difficulty poses any fundamental problem.

The procedure we follow to publish and retrieve data is nearly identical to the procedure defined in Section 6.3.1; we merely truncate all DNS names to remove the ".ws" TLD (i.e., we are querying for non-existent TLDs). During the publication phase, each of these queries eventually reaches a root ADNS. Our long and seemingly nonsensical strings do not match any TLDs defined by the root, therefore the root ADNS returns an "NXDOMAIN" response with the root's SOA record. Well-behaved RDNS cache each SOA record independently, keyed to the question that generated the response, even though each SOA record is technically identical (but for the TTL). The data retrieval process follows the same pattern as before as well—we simply record the TTL of the SOA record in the negative response.

6.5.2 Results

Next, we test the accuracy of our SOA TTL method with a procedure similar to that used to investigate the TTL method in Section 6.3.2. Again we attempt to publish a 32-bit message to each DNS server found. We re-use the scanning methodology from Section 6.2. We use roughly 90 PlanetLab nodes for this test. After each publication the node waits 10 seconds and then retrieves its message to evaluate the success of its publication attempt. Table 6.3 shows the results of these publication attempts. We encounter more difficulty pruning unusable servers during the scanning phase than in Section 6.3.2—19% of attempts take place at unusable servers as opposed to 15% for the TTL method. Most of these failures—18% of publication attempts—stem from a failure to identify bad TTL decrementing behavior.

With regard to the population of usable servers, we find the publication success rates to be similar between the TTL Method and the SOA TTL Method—78% and 79%, respectively. We find nearly twice the number of losses due to no data being found yet encounter fewer failures due to data corruption and packet loss in the SOA TTL method.

We use the same data lookup methodology as for the RD method in Section 6.3.2, except that we elide the lookups at 10 and 20 seconds—factoring in retries and delay, these lookups can in practice overlap. Figure 6.3 shows the results of our lookups as a function of time. We observe that total expungement of our records is the dominant source of error throughout the measurement period. Failures due to corruption (partial expungement) never exceed 4%, and as many as 8% of DNS servers disappear before the last measurement attempt. The overall success rate is comparable to the TTL Method through approximately 15 minutes, after which point the SOA TTL method degrades more quickly. At the end of the measurement period (128 minutes), the SOA TTL method maintains a 47% lookup success rate, as opposed to 67% for the TTL method. While our study does not extend to the 3 hour mark, we note that by default BIND will not cache negative answers for more than 3 hours[Zyt]. Therefore, we would expect a sharp drop in lookup success at 3 hours.

Though the SOA TTL method may exhibit a lower success rate than other methods, its success rate is sufficient such that a publisher may compensate by increasing the number of servers to which to publish a message. A competent publisher will publish to many DNS servers using any of our methods; the individual success rates of the the methods may be used to inform this parameter.

Pub. Attempts	89.5K
Unusable Servers	16.7K
Non-Responsive Servers	322
Non-Recursive Servers	199
Non-Decrementing TTL	16K
Weird TTL Decrementing	1.6K
Usable Servers	72.7K
Successful	57.5K
Failure: Packet Loss	3.3K
Failure: No Data Found	6.5K
Failure: Corrupt Data	5.3K

Table 6.3: SOA TTL Bit-Pipe publication results

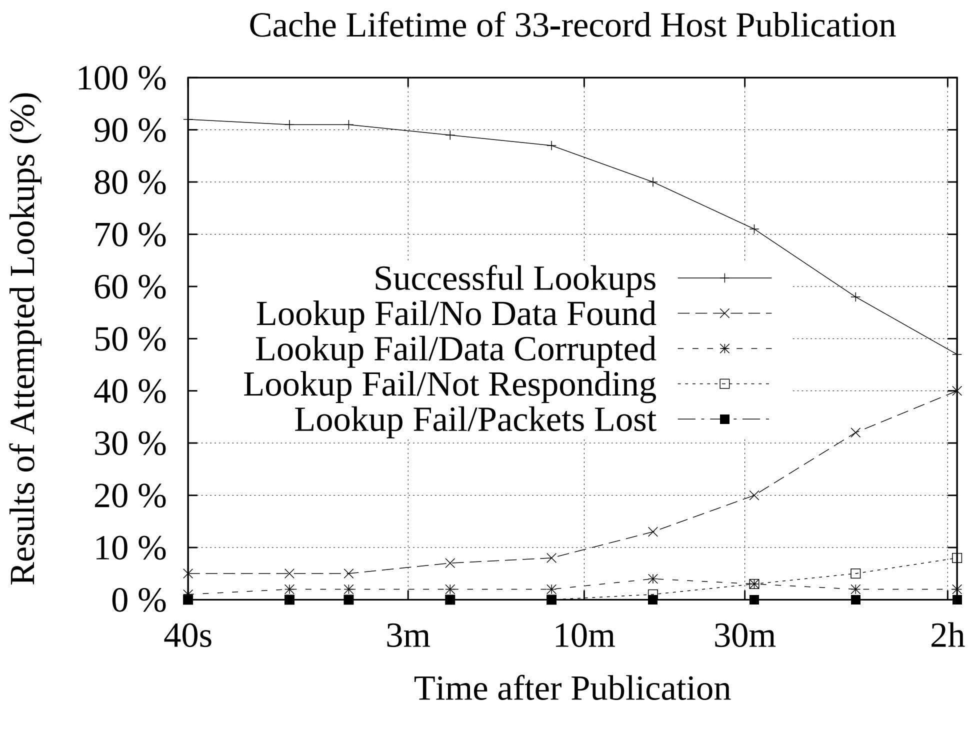


Figure 6.3: SOA TTL Bit-Pipe retrieval result

6.6 Additional Considerations

We now turn our attention to several additional issues surrounding the techniques discussed above.

6.6.1 Scalability

A naïve implementation of the methods we describe in this chapter will direct all communicating parties to a single DNS server, potentially causing problems due to load at the target DNS server. While our scheme fundamentally creates a channel for one-to-one or one-to-few communication, we note that this basic building block can be used to accomodate additional scenarios. In this section, we discuss a technique to build support for large numbers of communicating parties. We first discuss a many-to-many communication scenario for peer-to-peer network bootstrapping, and then we discuss a one-to-many scenario for disseminating information to many recipients.

In the peer-to-peer bootstrapping problem, our goal is to form all peers into an overlay network consisting of a single connected component. Our technique calls for a new parameter, $NETSIZE$, which is initially a guess at the number of peers trying to communicate. This guess dictates the size of the pool of DNS servers for peers to use. Each peer determines this parameter independently, however differing choices for $NETSIZE$ will not lead to distinct DNS server pools. That is, a peer with a $NETSIZE$ of 10 will have 5 DNS servers in common with a peer using a $NETSIZE$ of 5.

In order to obtain a $NETSIZE$ sufficiently close to the real number of peers in the network, we use an iterative process. Each peer begins with a large $NETSIZE$ and publishes its IP address to one server within the pool. After doing so, each peer randomly selects several other servers in the pool and attempts to retrieve peer IP addresses. If the $NETSIZE$ guess is too high, a low percentage of servers will contain messages; in effect, these random lookups will succeed with low probability. If the guess is too low, too many peers may use a single DNS server. We note that an inaccurate guess will not affect the correctness of this method, but rather its efficiency. In the case that $NETSIZE$ is too large, the peer cuts its estimate in half and tries again.

In each iteration, every peer picks a random integer N such that $0 < N < NETSIZE$. The peer then uses this integer to generate a "sub-secret", S', based upon the original secret S and the value N. Each peer then publishes its IP address using its S'. If $NETSIZE$ is sufficiently close to the true number of peers, then many of the N values within $NETSIZE$ will have been used. Peers will be able to guess the sub-secrets used by other peers and retrieve their IP addresses. As few peers will use any given sub-secret—which is then used to scan for a DNS server—the load on any particular DNS server will remain low. If a peer has made enough successful connections to other peers, the peer halts. We note that a single connection per-peer is insufficient as it is unlikely to form a single connected component in the overlay topology.

In addition to $NETSIZE$, we introduce two more parameters, which we discuss below:

- L: The number of lookups performed by each node in each iteration
- V: The number of lookups that must be successful for a node to halt

We now discuss an example of this process. In our first scenario, we begin with a pool of 10,000 peers who wish to form an overlay network consisting of a single connected component. In the beginning, no peer knows the IP address of any other peer. The process proceeds in phases, with each peer performing the following operations in each phase:

1. The peer begins with a guess of the number of all peers, $NETSIZE$, which we initially define as 10,000,000.
2. The peer chooses a random integer N such that $0 < N < NETSIZE$.
3. The peer constructs a sub-secret S' by appending N to the original secret.

4. The peer scans for an open DNS server and publishes its IP address using its sub-secret S'. If a publication already exists on the target DNS server, the peer retrieves the stored IP address and does not publish.

After Step 4, the peer has published its own address according to the randomly-chosen N value. In Steps 5-8, the peer attempts to retrieve data from several other peers.

5. The peer constructs L new sub-secrets by appending random N values (within $NETSIZE$) to the original secret.

6. The peer uses each sub-secret to scan for an open DNS server and retrieve data.

7. If the peer has not yet retrieved IP addresses using at least V of the sub-secrets, the peer reduces $NETSIZE$ by half and restarts the process at Step 2.

In order to test the potential of this method we perform a simulation. Our simulation consists of 10,000 nodes with a beginning $NETSIZE$ of 10,000,000. We use $L = 10$ lookups per iteration with a success threshold $V = 5$. We use the publication and lookup success rates observed for the TTL method, detailed in Section 6.3.2. As the duration over all iterations is less than the measurement duration in Section 6.3.2, we assume that publications stay in a DNS server's cache throughout the experiment (though subject to corruption). Each iteration lasts 5 minutes, with each node performing its publication randomly within the first minute of the iteration. If two nodes attempt to publish to the same DNS server within 10 seconds of each other, we classify the publication as corrupt. As our TTL Method publications only require approximately 10 seconds[8] to complete, a second publisher arriving after that time would not interfere. We further simulate all lookups as taking place at the end of each iteration, and report lookup success based upon the message's time-in-cache. We note that the parameter choices above, including initial $NETSIZE$, L, V, and iteration duration, would be provided by an application based upon application requirements. The values we chose work for a simple demonstration of the method.

[8]We recall from Section 6.3.1, the "barrier" record is surrounded by two 5 second gaps.

At every iteration of the simulation, we construct a graph where each node is a vertex and each successful retrieval establishes an edge from the retrieving node to the publishing node. We examine the largest connected component of this graph to determine the number of peers that would have successfully joined the network. Our simulation shows consistent results from run-to-run, with 99% of peers in a connected component by the 8th iteration. The graph is typically completely connected by iteration 10.

Next we assess the load on the devices involved. We first observe the load imposed on the open DNS servers. Across 10 runs of the simulation, the maximum number of packets received at any DNS server is 792. With respect to the peer nodes, each node requires 314 queries for each publication and/or lookup, which is the average number of probes before finding an open DNS server (281) plus the number of data queries (33). Each node performs one publication and 10 lookups per iteration, yielding 3,454 queries–an average rate of 11.5 queries per second over each 5 minute period.

We believe that neither 792 queries per server for open DNS servers or 11.5 queries per second for client devices are troublesome for modern networking devices. Furthermore, the parameters described above can be adjusted to yield a different node query rate.

We note that this technique is an initial proof-of-concept for enabling large-scale bootstrapping without overloading a single DNS server. This technique is not necessarily ideal, nor has it been optimized for any particular set of requirements. We leave refining this technique and optimizing it for particular applications as future work.

We next turn to one-to-many communication. We note that in many cases, the publisher may be able to deposit the message to several open DNS servers singlehandedly and thereby provide enough capacity for a large number of retrievers. For example, a 141-record "tweet"-style message as described in Section 6.4 may be retrieved from a single server over 500 times within a two hour period while maintaining an average rate of 10 queries per second. Further, a single publisher might utilize many sub-secrets as in the peer-to-peer example. Each sub-secret would add another DNS server as a replica of the message, multiplying the potential number of recipients. In order to spread query load, the recipients could use the $NETSIZE$ iteration process sketched above to find a random replica. Alternatively, the publisher could distribute the number of replicas along with the secret. A publisher capable of sending 10 queries per second could discover and publish on 229 replicas in a two-hour period, enabling over 114K lookups over the same time. As the author's own commodity residential Internet connection is capable of in excess of 1,000 queries per second, we believe one-to-many communication can be scaled effectively.

6.6.2 Eavesdropping

A third party who can monitor DNS queries and responses and understands the algorithms sketched above could perhaps decode the messages being transmitted. A simple way to prevent that is to *xor* the message with bits derived from the secret. E.g., for Twitter-like message transfer we could *xor* the low-order byte produced by $G("Bits\%d", i)$ with the i^{th} character of the message before publication. Then, the receiver can repeat this process with the data retrieved to obtain the original message. Even if an eavesdropper intercepts the message it will be meaningless without the secret used by $G()$.

6.6.3 Detection

Our mechanisms can be easy to detect by a network monitor due to scanning or unusual DNS traffic. This problem remains somewhat fundamental, but we offer several workarounds. First, per § 6.3.2, we can replace random scanning with hit list scanning, which both produces less volume and more valid-looking DNS traffic (i.e., little traffic to hosts that do not respond to DNS). Second, on the recipient side of the process scanning (in any fashion) might be needed only once, after which an actor could get bootstrapped into the system and then get a regularly updated list of servers to use via some other higher-bandwidth means.

There are several additional ways to obfuscate the activities described in this chapter: (i) messages could be split across servers, (ii) messages could be split across wildcard domains, (iii) follow-up traffic could be generated to make the DNS requests appear as more normal and productive and (iv) completely junk DNS requests could be sent into the system regularly to raise the noise floor and therefore make the requests used to exchange messages appear more normal.

The publication portion of the process will need at least some low-rate scanning to ensure information is being published where the recipients will find it. Even once a suitable server has been found and scanning is no longer required, publishing and refreshing the data in the system requires a large number of queries in a fairly tightly defined time window, which will be noticeable. Therefore, while some of the above obfuscation techniques may help it will be fundamentally difficult for a publisher to go undetected by a savvy network monitor.

6.6.4 Other Channels

An additional avenue for encoding information is to implicitly leverage DNS caching (both of valid records and invalid records [And98b]). We have found that often it is straightforward to determine whether a particular DNS server has a record cached or not simply by timing two requests. If the first takes much longer than the second then the first was likely a cache miss. Hence, testing for the presence or absence of records in a server's cache can be a basis to encode zeros and ones. This is akin to the RD method sketched in 6.3 but without the explicit use of the RD bit to query the cache directly. A timing-based cache presence scheme would share the binary-only property of the RD method and hence not be amenable to widening. The spirit of this technique can also be used with some content distribution networks (CDNs) where we can put objects with fairly arbitrary URLs in the web caches and then test for their presence (as described in [TAQR09]). There are some interesting properties of such a scheme (e.g., the message is essentially read-once given that requesting the records populates the cache). Additionally, timing issues could complicate the implementation. We leave a full treatment for future work.

6.7 Conclusions

This chapter discusses a way to communicate through the network without relying on fixed infrastructure at some central hub. This can be useful for bootstrapping loosely connected peer-to-peer systems, as well as for circumventing egregious policy-based blocking (i.e., censorship). Our techniques leverage the caching and aging properties of DNS records to create a covert channel of sorts that can be used to store ephemeral information. The only requirement imposed on the actors wishing to publish and/or retrieve this information is that they share a secret that only manifests outside the system and is never directly encoded within the network itself. Crucially we piece together a communication fabric from *classes of services* and not from particular instances of those classes. That is, while we require a recursive DNS server, we have shown that there are many DNS servers on the network that will suffice. While the process can be optimized by assuming the actors share more than just a simple secret—e.g., hit lists to scan in lieu of random scanning, or roughly synchronizing when messages will be published—we show in this work that these are not strictly necessary. We show that we are able to effectively use these channels for both bootstrapping information and for short messages (a la Twitter). Future work includes optimizing the process and looking for ways to make it more robust.

Chapter 7

Metadata Information Storage System

7.1 Introduction

The Internet looks very different today than it did 15–20 years ago. In particular, we highlight three trends. First, user-generated content has outgrown its humble beginning in the form of bulletin boards and has emerged as a significant presence in the Internet landscape. *Normal users* have become significant content generators alongside traditional content-providing services (e.g., news, weather and e-commerce sites). Among the Internet's most popular services are those that are mere frameworks to be populated with content as the users see fit (photo and video distribution, blogs, social networking services, etc.). Second, rather than simply sending email messages and looking up information, people are now using a myriad of applications to engage in rich interactions with each other, e.g., by sharing information (e.g., via photo or video sharing services), engaging in live communication (e.g., audio/video chats, instant message, multi-player games) and finding each other (e.g., social networking sites). Third, users access the network from an ever-increasing number and variety of devices and access points (e.g., computers at work and home, laptops in coffee shops, smartphones from anywhere).

The increased heterogeneity in terms of content providers, applications and access methods employed by users to obtain and provide information on the Internet is a testament to the power and flexibility of the basic Internet architecture. However, the situation has also created a large amount of user-specific meta-information that is required to access the actual content[1] such as URLs for photos, videos and blogs; buddy lists for instant messaging and voice-over-IP systems; cryptographic certificates; email addresses; relationship information; etc. This meta-information explosion is born out of necessity, but leads to a mess of formats and redundant data scattered across applications and devices.

Current applications cope with meta-information in two basic ways. First, applications generally cope with their own information reasonably well (e.g., email programs track email addresses, calendar tools track calendar subscriptions, etc.). Further, applications are sometimes given pair-wise capability to interact with each other (e.g., a social networking site bootstrapping friends from an address book application). Second, applications can utilize cloud computing platforms to aggregate meta-information at some central point in the network that is accessible from any connected device. This centralized approach removes the need individual devices have for information at the cost of trusting a third-party to hold users' data. Given these two suboptimal paths, we view the meta-information explosion as an opportunity to build a new primitive for the Internet that is tasked with coherently, comprehensively and securely dealing with arbitrary user meta-information.

In this chapter we describe the Meta-Information Storage System [CARB11] (*MISS*) which is a foundational service that provides users and applications with a new *architectural abstraction* for dealing with meta-information. The system can both deal with the current mess and more crucially enable new functionality based on wide-spread ready access to meta-information. We note that the actual system we describe in this chapter uses off-the-shelf components. Our contribution is not in designing some novel information store, but rather in the architecting of a service that offers a better way to organize meta-information within the network. Further, our goal is to design a "thin waist" sort of service

[1]For this work we consider "content" to be both stored information such as a blog entry, as well as ephemeral communications such as an audio chat.

(akin to the IP network protocol or the DNS naming system) that is at once general and simple, even if not always optimal. We view our re-use of previously developed mechanisms to be a positive comment on the community's effort to design generic technology that is broadly applicable. We view this work as representing another step in the process of creating generic, broadly useful services.

7.2 Meta-Information

The *MISS* system is tasked with dealing with *meta-information* which we define as a range of information that is not the ultimate content the user interacts with, but rather the information required to get at that content (names, social graphs, etc.). In the discussion below we describe *MISS* as both extensible and application-agnostic and therefore it is tantalizing to think about using the system to hold actual application content. However, we believe the system will ultimately be ill-suited for this use. First, as sketched above, we designed *MISS* with the idea of it being a "thin waist" for meta-information. Therefore, the interface is sufficient for that task, but likely not nearly rich enough to support the general content transactions. Second, the volume of meta-information in relation to content is incomparable. A system for storing a relatively small amount of meta-information is feasible, but a general content store represents a larger and more complex task.

The *MISS* data store calls for each user to have a *collection* of meta-information.[2] Each collection is comprised of a set of *records*, each of which is identified within the collection by a name and type. The *MISS* system itself is agnostic to what is held in these records—except for a few pieces of information needed by the system itself—leaving applications to craft records as needed. As an example to aid the reader's intuition, consider a record in Alice's collection containing her current email address. When Bob sends an email

[2]For ease of exposition we frame this work in terms of each user having *one* collection. However, there is no reason a user cannot have multiple collections within *MISS*—e.g., for different roles they might play (e.g., home vs. work).

to Alice, his email client can look her address up in Alice's collection at the time of email transmission. Whereas today's situation calls for Alice to tell Bob when she starts using a new email address, the late binding facilitated by *MISS* allows for Alice to arbitrarily change where her email is sent with no impact on Bob. This naming scheme is discussed in more detail in § 7.4.1.

Before describing collections and records in detail we sketch several requirements for the system. (1) The system should be *extensible* in terms of the types of meta-information stored to accommodate an evolving set of applications. (2) The system should allow the user to *control access* to their meta-information on a per-record basis (allowing for records to be private, public, shared within well-defined groups, etc.). (3) The system should provide for record *integrity* such that anyone with access to a record should be able to validate the record and that it belongs to the given collection.[3] (4) The system should be designed to allow for *portability* of meta-information with respect to where a user's information is hosted. I.e., collections should not be entangled with a particular service provider or meta-information hosting system.

Finally, we note that while *MISS* is charged with securely storing and sharing meta-information, that does not obviate the applications from conducting tasks such as access control (as they do now). For instance, a calendar available from a CalDAV server that is named within *MISS* (see § 7.4.1) will still require access controls at the server to limit who can access the calendar even if the record in *MISS* is encrypted such that only a small number of people can access the meta-information.

7.2.1 Collections

A collection is a container for all of a user's meta-information records. Collections do not have to be strictly tied to a single person, but could be used for organizations or groups,

[3]Related to integrity is trust. We do not require nor preclude a rigorous trust system. *MISS* can accommodate both externally developed trust, as well as applications that rely on looser notions of trust such as "opportunistic personas" [AKP+07] and "leap-of-faith" security [AN02].

however, for convenience we discuss collections as belonging to individuals. Each collection is associated with and named by a user-generated cryptographic key pair (C_s,C_p) for the secret and public halves, respectively. The collection's name is a fingerprint of C_p. By building *MISS* on a cryptographic foundation we address the bulk of the above requirements. Naming collections with user-generated cryptographic keys allows for portability as collections can be generated opportunistically by users and hosted with any willing service provider and moved at will. Further, the cryptographic foundation naturally allows both access control via encryption and validation via cryptographic signatures. We note that our *crucial assumption* for *MISS* is that the users' key material remain safe. When this assumption holds the system cannot deliver undetectable fraudulent records. In § 7.3.3 we discuss coping with the failure of this assumption.

7.2.2 Records

Each record within *MISS* is identified by a type, a name and the collection identifier. We place no constraints on the type and name fields, considering each to be an arbitrary string. The name of each record is chosen by the user (or by an application on a user's behalf). Each record is encoded in XML [BPSM$^+$00] both (i) in pursuit of the extensibility requirement and (ii) because libraries for dealing with XML are widely available for a large set of languages, allowing for easy integration into existing and new applications. With the exception of a few standard fields needed to track records, the system is agnostic to the contents of the records. This encoding strategy allows for a high degree of flexibility in the construction of the records. Further, record types can be added and changed as applications evolve.

Finally we note that each record in *MISS* has an associated expiration time which serves two purposes: (i) placing temporary records in the system can be done in such a way that no explicit cleanup is required and (ii) explicit expiration times allow for consumers to cache records to save querying time and expense on each use of a record (caching is discussed further in § 7.3.1). This mechanism is analogous to the time-to-live scheme used by DNS.

```
<miss_record>
  <name>myemail</name>
  <type>email-address</type>
  <expires>1278597127</expires>
  <signature> [...] </signature>
  <email-address>
    <ex1>alice@somenetwork.com</ex1>
  </email-address>
</miss_record>
```

Figure 7.1: Example *MISS* record.

Figure 7.1 shows a sample *MISS* record that defines Alice's current email address, per the example sketched above. The first few fields are standard to all *MISS* records and give the record's name ("myemail"), type ("email-address"), expiration time and the signature of the entire record. The collection that holds this record is not explicitly encoded in the record, but is needed to obtain and validate the record (as detailed in § 7.3.2). The remainder of the record (the "email-address" portion and enclosed address)is defined by an application and is irrelevant to *MISS*.

Since *MISS* is built on top of a cryptographic core the foundation for protecting users' privacy is in place and records can be encrypted such that only certain peers will be able to access the contents. While this allows users and applications to control the information placed into *MISS*, sometimes the mere presence of a record may violate a user's privacy. Therefore, encrypted records are only returned to users with access to decrypt the records.

We note that the *MISS* data model focuses on individual records and there is no index of the contents of a collection.[4] A requester must know what they are looking for (collection, record type and name) to form the request. This allows for record-level access control without carrying that forth to some form of index for enforcement. In addition, it offers a level of privacy in that while public records can be fished from the system,

[4]Of course, various forms of index records could be constructed for various purposes since *MISS* itself is agnostic to the contents of a record.

not readily publishing those records in an easy to find form prevents others from surfing through all of a user's meta-information. Finally, our focus on individual records keeps the interface between the meta-information servers and the clients simple (i.e., *get()/put()*) and only places minimal trust in the *MISS* servers themselves (see § 7.3 for a deeper discussion of the interface).

We also note that like transport protocol port numbers there are two general classes of record types: well-known and ad-hoc. The division allows for well-known rendezvous points and ad-hoc customized entries. Technically, the only difference between the types is that a registry[5] of well-known types and their corresponding record format will be kept (e.g., so two like applications can both understand the format). To avoid clashes in type names it is recommended that types expected to see broad use be registered. Additionally, ad-hoc types should be prepended with the name of the application to reduce the chances of collision (e.g., "Firefox-foo" instead of "foo"). Ultimately, however, applications must verify the record retrieved conforms to expectations and handle cases when it does not as an error.

7.3 System Overview

We next provide an overview of the system we have designed and prototyped to store users' meta-information. Figure 7.2 shows a high-level diagram of the system and will be described in the remainder of this section. We note that while we believe the design presented in this section is the best realization of the engineering tradeoffs involved, other designs were considered and are possible. The main contribution of this work is in the abstraction and not on the particulars of the system which could be engineered differently without impacting the provided foundation.

[5]While we have not yet set up a registry the history of IANA suggests that it is possible and works well.

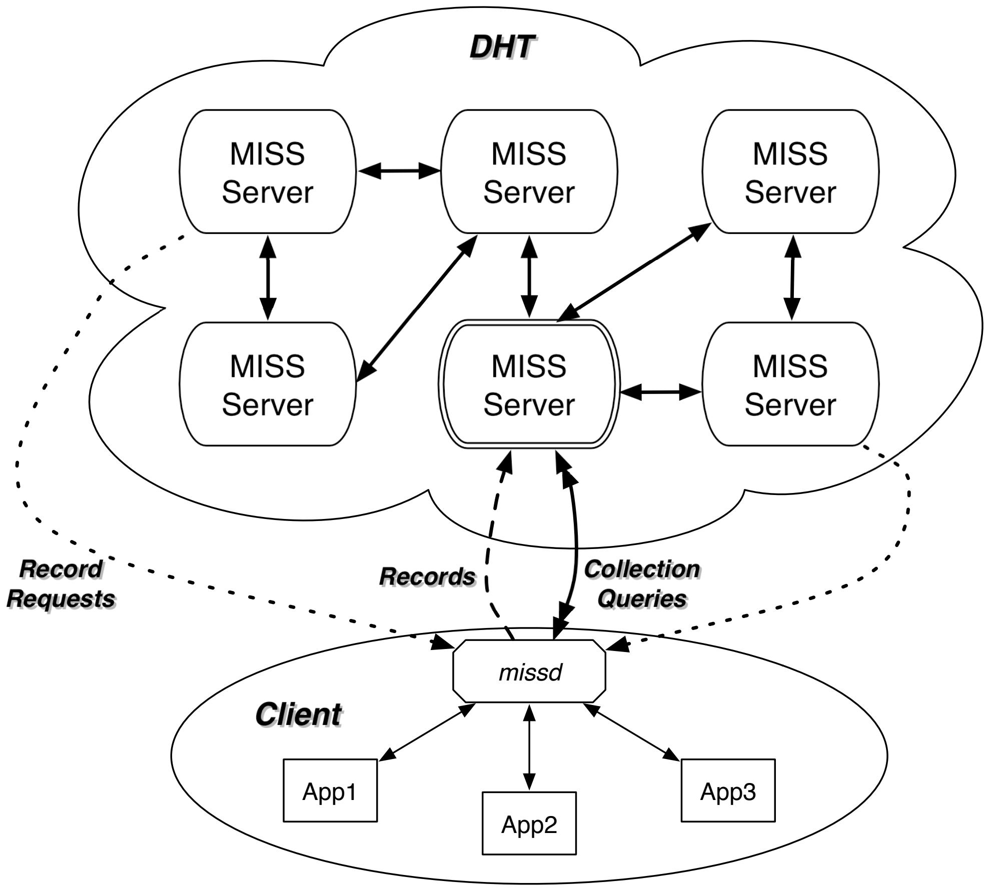


Figure 7.2: Conceptual diagram of *MISS* system.

7.3.1 Local Interface

The *MISS* system requires a local interface between applications and the global repository. As shown in Figure 7.2, this interface is implemented through a local service, *missd*, which runs on a user's behalf on the same device as the applications and provides several key functions and undertakes a number of common tasks, as follows. First, the *missd* provides applications with a general *put()/get()* interface to the meta-information in the overall system without requiring configuration and complexity for each application. Second, the *missd* holds the user's state (their own records, cached copies of others' records, etc.) and provides for meta-information management—e.g., as the user's records need to be added or updated in the global database (e.g., as expiration times near). Also, the *missd* handles common operations, such as signing and/or encrypting records, as necessary. In addition, the *missd* fetches other people's meta-information and can cache and pre-fetch these records to make access to often-used information quick.

We note that the simple *get()/put()* interface *missd* provides has proven sufficient for managing the meta-information in all applications we have worked through and/or implemented (see § 7.4). Therefore, to keep complexity low we have not constructed a richer interface based on hypothesized needs. That said, the interface may well need to be enlarged as needs arise.

7.3.2 Global Access

We now turn our attention to the structure of the global meta-information storage system. The system is comprised of the collection of *MISS* servers shown in the upper portion of Figure 7.2. These servers are presumed to be infrastructure-level nodes run by ISPs or institutions for their respective users (e.g., similar to email servers). Each collection of records is stored on at least one *MISS* server (and possibly several for robustness). Each *missd* is configured to know the *MISS* server(s)—denoted by the server outlined with a double-line in the figure—for its collections and stores its applications' records there. (The figure only shows one *MISS* server—i.e., no backup servers—associated with the *missd* for clarity.)

When querying the system for a record, the *missd* must first determine which *MISS* server holds the relevant collection. The *MISS* servers together form a distributed hash table (DHT) that stores a mapping of the collection identifier to the particular *MISS* server(s) on which the records are stored in a `master` record. Therefore, to retrieve a meta-information record the *missd* first queries the DHT with the collection identifier to retrieve the `master` record—as denoted by the double-arrowed line in Figure 7.2. An example `master` record is shown in Figure 7.3. The record starts with the four standard fields (name, type, expiration time and signature). Next, the collection information is enumerated, including the collection owner's name, email address and public key. Finally, there are a list of *MISS* servers which hold the records in the collection. Once the *MISS* server(s) for a given collection has been identified, the client queries one of the *MISS* server directly for particular records within the collection—shown with the two dotted lines on the figure. Each record is identified by a three tuple (c, t, n) where c is the collection identifier, t is the type of record and n is the name of the record. It is presumed the servers will be queried in the given order, however, to support load-balancing the record may include additional information to better direct the *missd*'s choice (e.g., an indication to choose a random server).

```
<miss_record>
  <name>collection</name>
  <type>miss</type>
  <expires>1278684469</expires>
  <signature> [...] </signature>
  <miss_collection>
    <owner>
      <name>Joe Smith</name>
      <email>jsmith@foo.com</email>
    </owner>
    <pub_key> [...] </pub_key>
    <server>128.1.2.3</server>
    <server>132.25.30.35</server>
    <server>68.45.100.7</server>
  </miss_collection>
</miss_record>
```

Figure 7.3: Example `master` record.

As noted, *MISS* naturally supports the use of multiple *MISS* servers for robustness. This is akin to DNS' notion of listing multiple authoritative servers for particular names. When publishing a record the *missd* attempts to publish the same record on each of the *MISS* servers the user employs (and retries failed *put()*s, as necessary). The *MISS* servers independently age information out based on the expiration time and therefore no additional coordination is required to remove stale information.

It is possible that a user updates a record on one *MISS* server but cannot reach another *MISS* server (e.g., a backup) at that moment, which creates an inconsistency. However, (i) the inconsistency will be fixed as soon as the *missd* can re-establish a connection with the unreachable *MISS* server and (ii) while the lagging *MISS* server may not provide queriers with the most recent meta-information, any returned records will not be expired and therefore should still be valid (e.g., the given record could be used from some *missd*'s cache at that point). This system of loose consistency has proven reasonable within DNS and therefore we expect it will also be reasonable for *MISS*.

Since *MISS* queries represent a new lookup layer that must be used before various application operations begin, we must consider the added delay in performing these lookups. To combat this extra delay, the system can heavily use caching and pre-fetching. We expect expiration times to be relatively long in many cases. E.g., recording a name for an email address (as discussed in § 7.2 and § 7.4) with an expiration of one week would allow for both reasonable email address changes and record caching to prevent users from constantly looking up a common correspondent's current email address. Further, for regularly used items (which *missd* can easily track since all interactions happen via this component) records can be pre-fetched as the expiration time approaches. Using caching and pre-fetching means that for common records there will be only negligible delay imposed by *MISS*. However, the delay for looking up less commonly used records will remain (see experiments outlined in § 7.6 for initial data on the extra delay). While we do not have a good estimate of how many records people will utilize, we note that if we assume records are stable and time-out every week, that means pre-fetching one record per minute would allow a *missd* to keep 10K records fresh.

Finally, we note that the system readily supports the portability requirement outlined in § 7.2, as moving a collection from one server to another merely involves altering the pointer to the *MISS* server(s) in the `master` record.

7.3.3 Managing Collections

We now delve into three issues the system must deal with in terms of managing the stored meta-information.

Bootstrapping:

The first problem *MISS* faces is a bootstrapping issue between users. To access another user's information (e.g., an email address as outlined in § 7.2) the user's collection's public key (or fingerprint thereof) is required. The collection ID can be transmitted in a number of ways that are already used to exchange contact information, e.g., email header fields, optional HTTP headers, meta-fields in HTML documents [KWGA09], email signatures, vCards, etc. In addition, we could setup well-known registries (built from *MISS* collections) such that people could map their collection ID to some human-understandable name in a well-known place. While the names would have to be unique within the registry, this is for bootstrapping only. After learning someone's collection ID it can be readily stored within one's own collection under some context-sensitive name (e.g., "Dad"). This allows for users to interact with the system in a natural way. As sketched in § 7.4.1 we have built a naming system within *MISS* that can handle such tasks.

Security:

Recall that in § 7.2 we noted our crucial assumption is that the key material associated with a collection not be compromised. As long as this assumption holds, the *information* retrieved from *MISS* can be verified as legitimate by the consumer. An attacker compromising components of the system (e.g., *MISS* servers) can *prevent* (or slow) record retrieval, but cannot provide undetectable fraudulent information given that the consumer must have the collection's legitimate public key to request information.

However, in reality *the key material will be compromised* on occasion. Once an attacker takes possession of the user's private key, records can be fraudulently inserted and modified which could have far-reaching implications (e.g., redirecting email (see § 7.2 and § 7.4.1) to an account controlled by the attacker). In the limit, users can recover from this breach by simply populating a new collection and informing their peers out-of-band about the new collection. This is an effective, but high cost solution. Therefore, we design a reasonable—although not fool-proof—procedure to mitigate such events without requiring a start-from-scratch approach.

We extend `master` records in three ways. First, updating a `master` record *adds* a new copy of the record rather than *replacing* the old version (and all non-expired versions are returned in a lookup), preventing an attacker from simply over-writing legitimate information placed in the system. Second, a `master` record can *redirect* the requester to a different `master` record to indicate that the collection is migrating to a new collection ID (i.e., from a compromised collection to a fresh and uncompromised collection). Third, a `master` record advertises a so-called *secondary public key* (SPK) at all times. The SPK of various collections will be cached in the consumer's *missd* as a matter of course when using the system (i.e., before a collection is compromised). A `master` record indicating a redirect to another collection must be signed by the private key corresponding to the well-advertised SPK. Since this key is only needed for constructing the redirect (a rare occasion), the private key can be kept offline (e.g., on a USB stick), making it harder to compromise. Use of the SPK prevents the attacker from transitioning to new key material for the user by inserting a fraudulent redirect. Meanwhile, the legitimate user will possess the private half of the SPK and therefore be able to publish a legitimate redirect to a new and uncompromised collection.

This scheme does not provide iron-clad guarantees. For instance, if the collection's primary and secondary key material are stored together and both compromised, there is little recourse besides starting from scratch—and even that may be fruitless if the attacker still controls a user's device (and hence can readily steal the new key material). An attacker that controls both the primary key and part of the *MISS* infrastructure, e.g., the DHT node holding the user's `master` record, can thwart legitimate key transition by blocking legitimate redirects to keep consumers in the dark. Another attack involves consumers with no prior use of a collection and hence no SPK on file. In this case the SPK itself can be undetectably forged. The presence of legitimate redirects can of course raise a red flag in this case. Overall we note that these key transition attacks require a confluence of several events and/or breaches across the system and therefore raise the bar for the attacker significantly. While we believe that further mitigation of these attacks is possible with additional layers of monitoring (e.g., charging *MISS* servers with announcing key transitions to users' out-of-band or tasking the *missd* with re-requesting `master` records to build a history if none exists), we leave these mechanisms to future work.

Sharing Collections:

We finally note that for a user to share their entire collection across multiple devices will require the *missd*'s state (e.g., records, collection IDs of peers the user interacts with, etc.) to be kept within the user's *MISS* collection.[6] Note: the state can be kept in an encrypted form such that no other user can access the data. In addition, the user's *MISS* key pair will have to be distributed to the various devices from which they will access *MISS*. In addition, updating *MISS* from more than one location incurs the risk of update collisions. Our focus on individual record operations enables us to use simple timestamp-based concurrency control. *MISS* returns a timestamp t with each *get()* and *put()* that indicates the timestamp of the current copy of the record on the *MISS* server. When updating a record, the t from the previous *get()* is included with the *put()*. If the corresponding record on the *MISS* server is newer than t an error is returned. The caller can then fetch the current record and update that as needed.[7]

[6]Alternatively a user could keep their state on a portable USB fob, access it via some networked filesystem, copy it to the local device via `scp`, etc. But, for ease of exposition and as a generally pragmatic path we assume the user utilizes *MISS* for this purpose.

[7]Note: like DNS, *MISS* is a "record store" and not a database. If an application requires multiple semantically linked records, then it must implement the appropriate concurrency control (which can likely be done within *MISS* given its extensibility).

7.4 Use Cases

We now turn to sketching several use cases for *MISS*. The first two (§ 7.4.1 and § 7.4.2) leverage *MISS* to provide fundamentally new functionality while the latter two (§ 7.4.3 and § 7.4.4) illustrate how *MISS* can help users more easily deal with current meta-information. We stress that the use cases sketched below are exemplars and, per § 7.4.5, we believe *MISS* has many additional uses.

7.4.1 Naming

Within the Internet many ad-hoc naming systems have been designed and implemented independently and generally provide application-specific names to uniquely identify a given resource. On the one hand, the ability to name resources as needed by applications in an ad-hoc manner is central to the general flexibility of the Internet architecture, and forcing all applications to use some rigid naming system would likely hinder innovation. On the other hand, the explosion of names, namespaces and artifacts of these makes the system difficult to grapple with for normal users. The two key problems with Internet names is a requirement for uniqueness and a tight coupling with a hosting provider.

At a basic level, names need to be unique such that one name cannot refer to multiple resources. This leads to names that are obtuse, ambiguous to users and difficult to share [FSLL$^+$06, SSV06, All07]. Consider the URL [BLCL$^+$94]. The URL may contain information about the application-layer protocol (e.g., “http”, “https”, “ftp”, etc.), the machine that hosts some resource (either a hostname or an IP address), the transport-layer port number, the filename on the server’s disk and/or arguments to some server-side program. The representation is inclusive and offers the flexibility to name a large number of resources, but at the cost of complexity. This complexity in turn makes the names hard for users to cope with directly and share. Further, global uniqueness leads to ambiguity for humans. E.g., does “ou.edu” embedded in a URL indicate the University of Oklahoma or Ohio University?[8] Since both of these institutions are referred to as “OU” in their respective regions the Internet name becomes ambiguous. In sum, while global uniqueness is required, forcing users to use such names comes with a complexity cost.

The second fundamental problem with our current namespaces is that names are tied to service providers [SSV06, All07]. Consider an email address. While a user may get some input on creating the username portion of their email address the hostname portion to the right of the “@”-sign denotes a location within the Internet where email will be delivered. Therefore, if a user wishes to change email providers they must garner a new address and inform their correspondents to use the new version. One problem with this scheme is that changing is costly enough that users are locked into service providers rather than having the real ability to choose providers based on merit, price, etc.

A previous work in this area proposed a *personal namespace* system that allows users to add *aliases* on top of the Internet’s myriad namespaces, hence creating a meta-naming layer that is better suited to human use [All07]. While that proposal envisioned a system for storing aliases, the *MISS* system is essentially a superset of that system. Two example personal namespaces from [All07]—for Alice and her son Bob—are given in Figure 7.4.[9][10] Each record in the namespace has a type given in **bold**, a name given in *italics* and a value given in normal text which is the standard unique name for the given resource. Alice controls the names for her own resources, e.g., by assigning her current email address to “myemail”. Her son can set a pointer to Alice’s namespace using the context-sensitive name “Mom”. He can further set a pointer to her email address as “Mom:myemail” which will resolve to Alice’s current email address. Alice can change her email address—because she wishes to switch providers, gets a new job, etc.—without any impact to Bob who continues to simply use “Mom:myemail”. We also note that Bob’s namespace illustrates the use of *MISS* to track other users’ cryptic collection identifiers, as sketched in § 7.3.3, in his

[8]The University of Oklahoma

[9]Figure used with permission.

[10]Note: The format used here is for illustrative purposes. As mentioned previously the information would be encoded in XML for storage within *MISS*.

Alice's Collection:

email *myemail* = alice@mailserver.com
rss *blog* = http://blogserver/alice-blog.xml

Bob's Collection:

pointer *Mom* = [Alice's Collection ID]
email *Mom* = Mom:myemail
URI *vacation07-pix* = http://www.flickr.com/[...]

Figure 7.4: Example personal namespaces.

pointer to Alice's collection.

Using *MISS* to allow users to alias their own resources addresses both fundamental problems with Internet names. I.e., by allowing users to alias unique names we allow for human-friendly, context-sensitive identifiers that are easy to use and share. Additionally, we break the coupling between names and providers by allowing people to operate on aliases that are controlled by users within their *MISS* collections.

Together with our prototype *MISS* system (see § 7.6) we have implemented this naming system within the Firefox web browser and the Thunderbird email client via plugins (publicly available from [ucn]). Once enabled this allows users to access web pages and send email using aliases stored in *MISS* that resolve to traditional URLs and email addresses.

7.4.2 User-Directed Protocols

We next turn our attention to using *MISS* to allow users to more directly control the flow of their transactions through the network. The naming mechanism discussed in the last section provides one aspect of this by decoupling names from service providers and by allowing users the flexibility to quickly change providers. However, such wholesale changes of providers represent only the beginning of the ways a general meta-information storage system can be leveraged to allow users to control information flow. Below we sketch two different ways to employ *MISS* records to give users more fine-grained control.

Adding Redundancy:

The first use of *MISS* to allow users to direct traffic involves the user explicitly requesting redundant delivery of data. While we discuss this in the context of email, the general concept of specifying redundant delivery is a more widely applicable notion.

While the Internet's email system generally works well there are two fundamental problems in the overall mechanism. First, email has no end-to-end delivery guarantee, but rather the system is composed of a succession of SMTP servers that are each a potential point of failure. Further, there is no over-arching recovery process to detect and/or recover from delivery failures. Second, each point in the process attempts to deliver each email message even when the next hop is unavailable and this can lead to prolonged delays in message delivery. For instance, consider the case when an institution's email server loses connectivity due to a prolonged power outage. Mail destined for the institution will either be queued up in intermediate nodes and retried periodically or be sent to a backup server as defined in the DNS. Since messages at the backup server are not likely available to users (via IMAP, say) both of these situations leave messages in limbo until power is restored to the recipient's institution.

There are proposals for solving the first problem. In fact, the SMTP specifications call for "delivery status notifications" [Moo03] and "message disposition notifications" [HV04]. However, the system is still largely implemented using hop-by-hop reliability instead of end-to-end reliability. That is, once a message has been handed off to the next hop, a mail server no longer tracks it in any way. Alternatively, SureMail [APJ05] is a system whereby the sender deposits notations in a DHT that indicate a message was sent to a given recipient. Only the recipient can retrieve and understand these notations. Any email that has been noted as sent but has not been received can be requested from the sender. The second problem of prolonged delivery times has never been addressed to the best of our knowledge.

A straightforward use of *MISS* can mitigate both issues. As sketched in § 7.4.1

users can give their email addresses aliases within *MISS*. Rather than setting this address to a single address the user can set it to multiple email addresses separated by commas. Such a string will be accepted by the vast majority of email clients with the message being sent to both addresses. In this way a user can force the system to redundantly deliver their email[11] and hence mitigate the reliability and timeliness issues discussed above. Our prototype supports this use of *MISS*.

Hooks:

Above we sketched aliasing to facilitate choice in providers and redundancy to increase robustness and timeliness as mechanisms users can employ within *MISS* to control their transactions. In this section we specify more fine-grained control. Much like CLISP [cli] provides "hooks" that are run before or after some activity to customize the process we propose providing such hooks in the Internet. *MISS* provides a natural place for users to set such hooks to inform the system about a user's preferred delivery process. We continue with email as our exemplar. An institutional SMTP server may consult a "pre-delivery" hook in the recipient's *MISS* collection before delivering a message to the IMAP server. In this record the user could specify a spam checking service to which the incoming message is submitted and the returned message is delivered to the user (say, after adding headers with the checker's results). This added flexibility gives users choice in terms of picking the best possible technologies rather than a bundle of technologies that is put together by a given provider. While in some sense users can do this now (e.g., by running their own spam filter) that is (i) often an impractical path for normal users and (ii) sometimes beyond the abilities of a given device (e.g., a smartphone).

In addition to email filtering, hooks could be used to specify services for web proxies that remove unwanted content (e.g., malware or ads). Alternatively a user on a smartphone could direct traffic through a transformation service that reduces picture quality, increases

[11]Note, email clients can effectively use the Message-ID email header to remove duplicate messages so the user is only presented with one copy of each message.

font size, etc. for both better viewing on a small device and better use of the scarce network resources often encountered on a 3G link (say). Another use might be a service that could observe incoming messages (email, IM, etc.), detect particularly important messages and send the user a notification to their phone (e.g., via SMS). We believe that as is the case in programming languages hooks could be a powerful concept in giving users a mechanism to flexibly control network transactions.

7.4.3 Sharing State

As sketched in § 7.1, the devices people use to access Internet services are increasing in number and variety. It would not be atypical for a user to access the Internet via multiple devices of differing capabilities in the course of a day (e.g., desktops, laptops, tablets, smartphones, etc.). One of the consequences of this access pattern is at any given time the user does not have access to all their application state. For instance, setting a bookmark in Firefox at the office does the user no good when later accessing the web via their tablet. Likewise, finding and pulling up a web page on one computer is of no use if the user only has time to read the page when they are away from that computer but have access to their smartphone. Ad-hoc solutions exist for various applications (e.g., storing bookmarks in the cloud). However, *MISS* provides a framework for addressing the problem in a coherent and comprehensive fashion. Since applications can easily push custom records into *MISS* they can readily save bits of state in the user's own collection for retrieval from another device. E.g., an RSS reader could record the article IDs associated with articles the user has read such that they do not appear as "new" when a different device presents the list of articles to the user.

We prototyped another example application on top of the *MISS* system discussed in § 7.6. We have built a Chrome extension and Firefox plugin to save and retrieve the state of a user's browser in *MISS* (both extensions are available at [ucn]). The plugin takes a snapshot of the URL associated with each window and tab open at a given time[12] and saves this in the user's *MISS* collection as a record that only the given user can access (i.e., decrypt). When the plugin is asked to retrieve the state it queries *MISS* for the given state record and opens windows and tabs as necessarily and loads the corresponding URLs. Exactly what gets stored in such records is determined by the applications. For instance, while we have not yet done so, we could store each tab's history, as well—allowing the user to use the "back" button on the restored state just as they would have on the origin browser. Again, existing solutions such as remote desktops can achieve similar functionality but *MISS* can support it as part of a foundational service.

While the previous two use cases of *MISS* have been aimed at utilizing the *MISS* data store to provide new functionality this use case is a bit more mundane. For a number of applications there are in fact existing ad-hoc mechanisms to share state between devices (e.g., bookmarks, address books, RSS aggregators, etc.). This use case is therefore an illustration that *MISS* can be used for current applications, as well. However, even for existing applications there are benefits in a standard interface (e.g., we can share current browser state across Chrome and Firefox). Further, applications can leverage the *missd* to handle common tasks and therefore do not have to be nearly as complex as point solutions to accomplish the same task.

7.4.4 Storing Configuration

Similar to storing application state across devices and sessions, *MISS* can be used to store configuration information. Such configuration can be complex and therefore cumbersome to deal with. E.g., consider the configuration for a typical email client. For each incoming email account the user accesses the client will require the hostname of an IMAP server, the TCP port on which the server is listening, a username, a password, possibly an SSL

[12]Currently the user is required to push a button to save their state, but we could trivially make the browser save the state on particular events.

certificate, configuration options (e.g., whether to keep mail on the server or not), etc. In addition, to handle outgoing email the client will also need the name of an SMTP relay, TCP port number and possibly credentials to access the relay. This information needs to be configured into every client a user employs to check their email (e.g., at least once per device they use and each configuration change must be manually propagated to all devices). Similar configuration information is required within many applications (e.g., instant messaging setup, HTTP proxy settings, simple general preferences in myriad tools, etc.).

MISS provides an opportunity to encode this configuration information once and then retrieve it for use on various clients. As with storing application state discussed in the last section, this use of *MISS* moves towards a more seamless integration of devices. I.e., no longer is it a manual burden to setup a new device or access information from a temporary location. In addition, once applications understand how to retrieve configuration information from a standard *MISS* record this will pave the way to remove even more manual configuration by allowing institutions to provide most of the information via a institutional collection. The users would then only have to deal with pointing their application at the institutions configuration record (using some canonical name, say) and conducting user-specific configuration (e.g., credentials). And, this latter only needs done once and then can be used across clients and devices. Such a system provides flexibility for an institution in terms of changing the configuration without requiring manual intervention by users or system administrators on each application instance.

As with the use case described in § 7.4.3 this application of *MISS* is not a fundamentally new capability. Rather, this use case illustrates the ability of *MISS* to help uses cope with a current source of meta-information. Once a system such as *MISS* is in place many incremental benefits such as the examples we have discussed are likely to be straightforward. While not necessarily compelling in their own right, we believe the sum of such benefits will provide a qualitatively better system.

We have developed a Thunderbird plugin to retrieve email client configuration information from our *MISS* prototype (see § 7.6) and then use that information to configure the application. Our plugin is available from [ucn].

7.4.5 Other Uses

We stress that the use cases developed above are illustrative examples that serve to demonstrate the potential benefits of a system to manage meta-information. We envision additional uses, such as:

- Several proposals have advocated removing social network data from social networking sites like Facebook (e.g., Authenticatr [RF08]). *MISS* would be a natural place to store users' social graphs such that they would be available across applications.
- We have previously proposed the use of "assistants" to which small housekeeping tasks can be delegated such that a host can enter a power-saving sleep mode without losing its standing in the network [ACNP07]. Such delegation information could be conveyed within the *MISS* system in a secure fashion.
- *MISS* could be used to encode mirrors to particular data. This could be done to load balance the transferring of some large file (e.g., a Linux distribution) from multiple large servers. Or, it could be the basis of some private file trading service among a small, private group that does not possess fixed infrastructure.

There are no doubt many more examples. Our contribution is in a flexible and extensible foundation that can enable such uses—including those we cannot yet envision.

7.5 Incentives

We briefly consider the incentives to use and deploy *MISS* for three directly impacted constituencies here.

7.5.1 Users

The entire design of *MISS* revolves around providing users both ease-of-use and flexibility in dealing with networked systems. As sketched in § 7.4 the uses of *MISS* from the user's point-of-view are numerous and therefore we believe an instantiation of *MISS* that mostly hides the system complexity will have immediate appeal.

7.5.2 Developers

The incentive for developers to use *MISS* in their applications is two-fold. First, there is a natural incentive to add value for users and to the extent that users benefit from *MISS* then developers will be incentivized to use the system. Second, applications naturally have meta-information management needs and so leveraging a general system rather than building and maintaining an ad-hoc mechanism is appealing. This latter may not hold as strongly for existing applications that already implement an ad-hoc mechanism, however, migrating to a general framework that allows for both removal of complexity for the application and benefit for the user is still an incentive.

7.5.3 Operators

Finally, system administrators and network operators—at ISPs and institutions—will be required to run *MISS* servers. *MISS* may well help their own processes. However, the largest appeal of *MISS* will be to add value for their users. There is obviously a track record for such services to be offered to users (e.g., email, web space, etc.).

None of this is meant to suggest uptake of a system like *MISS* will happen overnight. There is still a certain chicken-and-egg property to the system. Our goal in this section is to note that there are tangible incentives for the use and deployment of a *MISS*-like system.

7.6 Experiments

We have prototyped the *MISS* system, including the *missd*, *MISS* server, DHT functionality and several application plugins (as sketched in § 7.4). Communication between the various components of the system is handled via XML-RPC[13]. We use the Bamboo DHT [Bam] system to connect the *MISS* servers. Our prototype implementation and the plugins discussed above are publicly available [ucn].

A correctly working prototype is just a beginning, however. Since storing meta-information in *MISS* results in an added lookup time during application processing we now turn to an initial evaluation of the performance of our prototype. Our experiments concentrate on three aspects of users retrieving information from *MISS*: the capacity of a stand-alone *MISS* server, end-to-end tests of the system across a small LAN testbed to better understand the non-networking costs and finally end-to-end tests across a more realistic wide-area deployment of *MISS* to understand the networking costs. While we briefly touch on record insertion below, our focus is on information retrieval as we believe that will be the most prevalent interaction with the system and information insertion will be relatively rare and can happen behind the scenes and not hinder users' activities.

We note that the workload imposed in the following experiments is in some ways contrived. However, in the absence of a production meta-information system we are not able to leverage a "typical" workload. There are aspects of such a realistic workload that will no doubt matter to performance (e.g., collection popularity, locality, etc.). However, since there is no realistic model of this new system our goal in this section is to get an initial understanding of the order of the imposed delays. In our future work we will endeavor to refine these experiments based on more realistic usage patterns as we involve actual users in our experiments.

[13]http://www.xmlrpc.org

Concurrency	**Sustained Req. rate (K/sec)**	**Avg. Resp. Time (msec)**
1	1.7	< 1
500	27	18
3100	5.6	550

Table 7.1: Stress-testing of a MISS server.

7.6.1 MISS Server Load

Our first experiment aims to assess whether a reasonable server can handle a query load from reasonable-sized organization. We setup a *MISS* server running Apache 2.2.14 and used a client machine running ApacheBench 2.3[14] to stress test the server. Both machines were configured with Ubuntu 10.04 Server, each having two quad-core 2.40GHz processors with 8GB of RAM. We varied the load using ApacheBench's concurrency setting. Table 7.1 shows our results. An unloaded server—answering one query at a time—can handle the requests in under 1 msec. Increasing the concurrency to 500, our *MISS* server can support over 27K requests/second with an average response time of 18 msec. However, further increasing the concurrency to 3,100 yields both a longer delay—550 msec—and a lower aggregate service rate (5.6K requests/second).

Obviously in the absence of an operational deployment we do not have estimates of a reasonable query rate. However, as a rough approximation we note that at the Lawrence Berkeley National Laboratory (LBNL) one of the main DNS servers received an average of 121 requests/second over the course of one day in July 2010 [PS10]. While there is not an exact equivalence between DNS lookups and our expectation of *MISS* lookups, the DNS load is suggestive of user activity levels in that web page retrievals, email transmissions, etc. trigger name lookups. Taken together the experiments and the LBNL data suggest that a single reasonable server-class host could handle the load imposed by a large organization without imposing significant additional delays to the process.

[14] http://httpd.apache.org/docs/2.0/programs/ab.html

Operation	**Median (msec)**	**95th perc. (msec)**
Parse/verify	22	26
`master` fetch	3	6
Record fetch	2	3

Table 7.2: Overhead of *MISS*.

7.6.2 Local Network Baseline

Next we turn our attention to assessing the overhead of the basic *MISS* system we have built using a small LAN-based testbed. We deploy three *MISS* servers which also form a DHT. Each *MISS* server holds 100 collections of 100 records each. We next configure three clients to each fetch 30K random records. Each retrieval involves first obtaining, parsing and validating a `master` record followed by retrieving, parsing and validating the actual meta-information desired from the given *MISS* server. The results are in Table 7.2. The entire process finished in at most 35 msec for 95% of the cases. We note that the client operations of parsing and verification—not the networking operations—dominate the time required in this environment. This experiment shows that obtaining local data—which we expect to be a common occurrence due to people's general habits which tend to have a local focus—is not particularly time consuming compared with many common networking operations (e.g., loading a web page).

In the same three-node setup we also tested the naming plugin we developed for Firefox (sketched in § 7.4.1). Within Firefox we retrieved ten records from each of three collections hosted on the testbed. Over the 30 tests we find a median retrieval time of 31 msec or 4 msec longer than we find above when using an automated retrieval tool and not an actual application. We therefore conclude that our experiments are useful predictors of real application performance.

7.6.3 Internet-Wide Test

Our final set of experiments involve assessing the time required to fetch meta-information across the Internet within *MISS*. Our experiments are conservative since they do not take into account caching and prefetching as discussed previously. To understand how such mechanisms impact performance we would need a model for user activity that—in the absence of a production meta-information system—we do not have.

For this experiment we picked 100 random PlanetLab nodes to act as *MISS* servers and placed 10 collections with 100 records each on each node. The *MISS* servers form the DHT to provide `master` records. Our first set of experiments involve using client machines at ICSI and Case to retrieve meta-information. For each measurement we (i) choose a record at random from our corpus, (ii) download the associated `master` record from the DHT starting with a random DHT entry point[15] and (iii) fetch the given record from the *MISS* server indicated in the retrieved `master` record.

We retrieved 19K records using the ICSI client and 34K records using the Case client. Figure 7.5 shows the distribution of the retrieval time to the Case client. The distribution from the ICSI client is similar—with times being slightly less. The results show that the total time to fully retrieve a piece of meta-information is over 1 second in nearly 60% of the cases across both datasets. Further, 10% of the retrievals take more than 2 seconds to ICSI and more than 2.5 seconds to Case. The non-networking components of the delay are similar to that shown in the last set of experiments in absolute terms—which is much less as a fraction of the entire process for the wide-area experiments. Finally, we find that the two components of the retrieval process—fetching the `master` record and retrieving the meta-information record itself—take roughly the same amount of time (over 0.5 seconds at median). These results show that meta-information retrieval in the system we have designed consumes a non-trivial amount of time relative to the duration of normal

[15]Using a random DHT entry point is conservative and may impose added delay over a situation where the user can simply use a local *MISS* server for this purpose.

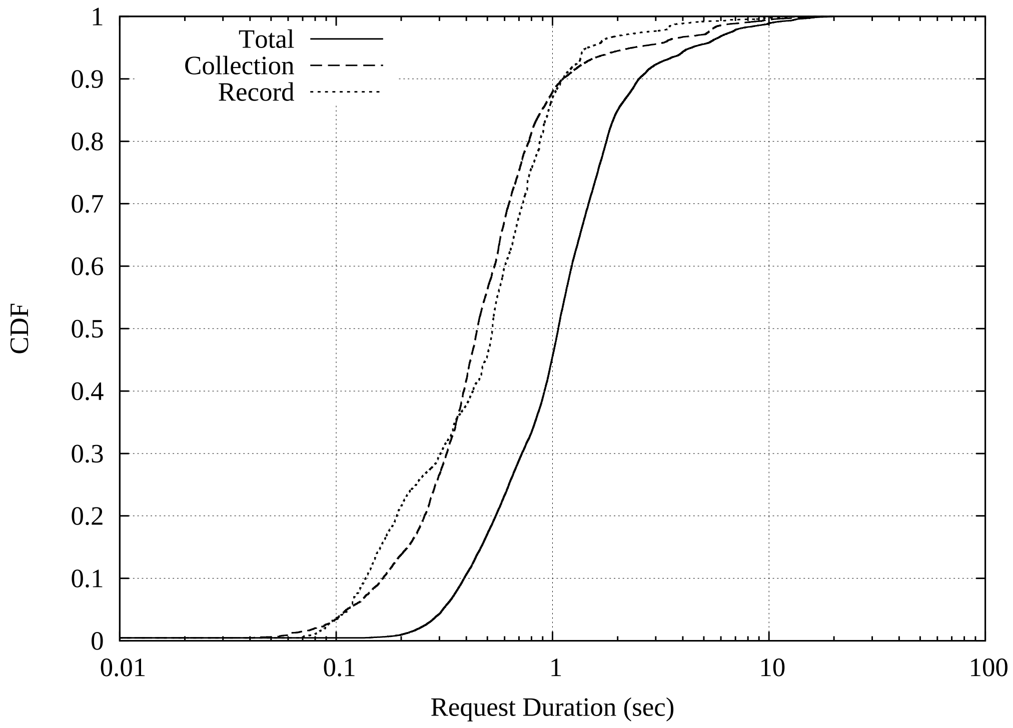


Figure 7.5: Duration of information retrieval from *MISS* to client at Case.

Internet transactions. This is especially so if the meta-information retrieval is part of a task whereby the user is actively waiting on the results. The retrieval times suggest that caching and pre-fetching of commonly used records will be crucial.

7.6.4 Record Insertion

Finally, we assessed inserting information into *MISS* in the context of our PlanetLab experiments. We find that inserting `master` records into the DHT takes a median of 831 msec (with the 95^{th} percentile being 9.2 seconds) and inserting meta-information records to particular *MISS* servers takes a median of 386 msec (with the 95^{th} percentile being 1.2 seconds). The former time is of little concern because inserting `master` records is a rare occurrence. The latter time is likely a dramatic over-estimate of reality since we expect in general users will be inserting records into close *MISS* servers and our experiment utilizes random nodes throughout PlanetLab. While long, we expect these delays will largely be "behind the scenes" and therefore they will not impede users.

7.7 Conclusions

We make several contributions with this work. First, we describe a new architectural foundation to hold users' meta-information. The system, *MISS*, accommodates arbitrary application-defined meta-information. We build the system on top of a cryptographic foundation such that access control and data integrity are built in at the root of the system. Finally, the *MISS* system allows for portability of meta-information—i.e., a user's meta-information is not tied to any particular server or provider and can be easily moved between hosting services. This gives users flexibility and choices as their situations change. We have sketched a variety of use cases for the system and also illustrate the performance of a prototype we developed. While ultimately it is difficult to try to assess an undeployed architectural framework, we believe the system sketched herein offers potential benefits and also that it continues the community's conversation on user-centric technologies. This work represents only an initial foray into the space. Our future work on *MISS* will revolve around deploying test setups and integrating applications into the system such that we can then get the system into users' hands in a non-trivial way. This will then allow us to better understand how *MISS* will be used and any limitations that require better system design.

Chapter 8

Summary

In this dissertation we study four distinct aspects of Internet naming. In Chapters 4 and 5, we examine the usage and behavior of the DNS with respect to its originally intended purpose—performing and caching name lookups. In Chapter 6, we present a novel communication technique that leverages the DNS to both rendezvous and exchange small amounts of data. Finally, in Chapter 7, we discuss the design, prototype, and evaluation of a new user-centric naming layer for the Internet.

In our active measurement study of DNS we assess over 1 million Internet-connected DNS devices in order to evaluate the topology, caching behavior, security, and other behavior encountered by users of the DNS. We find that a majority of these open servers are in fact home devices that forward lookups on to one or more ISP-level resolvers. We observe a high churn rate with respect to these IPs; over 40% of servers disappear within 3 days. Many of these home devices modify the TTLs of records—especially records with a TTL of 10 seconds or less—indicating that system designers cannot rely on DNS TTLs to stop traffic to IPs no longer in use. Finally, we find that over 10% of devices that perform recursion remain vulnerable to a serious DNS security flaw [Kam08b].

Our passive study of DNS examines 14 months of DNS traffic from a small residential network. In this study, we investigate the requests made by users, the responses received, and the action (if any) taken by the user based on the response. One notable finding is that in 93% of cases where a DNS response contains more than one server, the servers all appear to be in the same geographic location. This indicates that these multi-answer packets generally aim to provide load balancing, as a solution designed for robustness would likely include diversity in network paths. We find that 40% of DNS responses in our dataset go unused by clients, suggesting that DNS prefetching may be at play. We bolster our finding from our active measurements that systems administrators cannot rely on TTL expiration to stop traffic as we observe that clients themselves will use DNS records even days after expiration. Finally, we find that clients disproportionally use the first answer in DNS responses—doing so roughly 70% of the time, regardless of how many answers are in a response.

In Chapter 6, we develop a novel communication scheme. To our knowledge, this is the first technique suitable for passing messages between computers on the Internet that needs no fixed infrastructure or pre-arranged rendezvous point. A core use case of this system is to provide such a rendezvous service to other applications. In this work, we develop a methodology for two parties sharing the same algorithm to scan an identical sequence of Internet IP addresses for open DNS servers. Further, we provide several methods by which parties can read and write a small amount of arbitrary information into the cache of a recursive DNS server. We experimentally evaluate the success rate for both write operations and read operations as a function of time. We evaluate these techniques using several hundred thousand open DNS servers. We were able to successfully convey IP addresses and 140-character ASCII strings via DNS server caches and reliably retrieve them over the course of several hours.

For our final piece of work, we develop a new Internet naming layer focused around users. In Chapter 7, we describe a meta-information disaster; that is, information describing content or how to get to it is difficult to share and/or control. For instance, email applications do not share contacts with IM clients, configuration data is not regularly shared across devices, and many identifiers (such as the URL to a friend's website) are obtuse and and onerous to transmit out-of-band. To mitigate this mess, we designed *MISS*, a system to manage, store, and access user meta-information in a secure manner. We base the system on a cryptographic foundation to provide integrity and allow for straightforward access control. Applications can store arbitrary data in *MISS* encoded in XML, allowing easy implementation yet providing flexibility. We also design *MISS* to be portable—allowing a user's information to be easily migrated among providers at the user's sole discretion. We discuss several use cases for *MISS*, including personal namespaces, which allow a user to add aliases on top of the Internet's current naming systems, allowing users to create meaningful identifiers for content. Furthermore, as each namespace is user-specific, standard names would allow some common types of a user's content to be found automatically. We described configuration sharing, whereby all of a user's devices could share application state. Finally, we implement a prototype system and assess its performance, finding that a commodity server could handle the metadata service needs for a reasonably-sized organization.

Appendices

Appendix A

NXDOMAIN Hijacking ISPs

Alice DSL, Bell Mobility, Bharti Broadband, Brasil Telecom S/A - Filial Distrito Federal, Cablevision Systems Corp., Cavalier Telephone, CenturyTel Internet Holdings, CHARTER COMMUNICATIONS, China Internet Network Information Center, China Mobile Communications Corporation, CHINANET Anhui province network, ChinaNet Guangdong Province Network, CHINANET Guangxi province network, CHINANET Hainan province network, CHINANET henan province network, CHINANET Hubei province network, CHINANET Hunan province network, CHINANET jiangsu province network, CHINANET JILIN PROVINCE NETWORK, CHINANET neimeng province network, CHINANET network, CHINANET ningxia province network, ChinaNet Shanghai Province Network, CHINANET Shanxi(SN) province network, China Network Communications Group Corporation, CHINANET Yunnan province network, CHINANET Zhejiang province network, CNC Group AnHui province network, CNCGROUP Beijing Province Network, CNC Group Chongqing province network, CNCGroup FuJian province network, CNC Group Gansu province network, CNC Group Guangdong province network, CNC Group Guangxi province network, CNC Group Hainan province network, CNC Group Hebei province network, CNCGROUP Hebei province network, CNCGROUP Heilongjiang province network, CNCGROUP Henan province network, CNCGROUP HuBei Province Network, CNCGROUP IP network, CNCGROUP IP network, CNC Group Jiangsu province network, CNC Group

JILIN province network, CNCGROUP Jilin province network, CNCGROUP Neimeng Province Network, CNCGroup Shan1xi province network, CNCGROUP Shandong province network, CNC Group Shannxi province network, CNCGROUP Shannxi Province Network, CNCGROUP Shanxi province network, CNC Group SiChuan province network, CNCGROUP Tianjin province network, CNC Group Yunnan province network, CNC Group Zhejiang province network, CNCGROUP Zhejiang Province Network, COMPANHIA DE TELECOM. DO BRASIL CENTRAL, Computer Sciences Corporation, Conversent Communications, Cox Communications, Data Communication Division, Deutsche Telekom AG, EarthLink, Embarq Corporation, Extend Enterprises Limited, Frontier Communications, GTE.net LLC, Hutchison 3G UK Limited, Interbusiness, IUnet/IT, Jazz Telecom S.A., Jiangxi provincial network of China Netcom, KBS Internet, Wholesale ISP/DSL Provider, KRNIC, Moztel LDA, NET SITE S.A, New Edge Networks, OJSC Ingush-electrosvyaz, Oman, ONO, OpenDNS, LLC, OPTUS INTERNET - RETAIL, PCCW Limited, Qwest Communications, RNC, Road Runner, Rogers Cable, Shaw Communications, SHYAM INTERNET SERVICES PVT LTD, Slovak Telecom, a. s., StarHub Cable Vision Ltd, TDKOM INFORMATICA LTDA., TDS TELECOM, TE-Data & IDSC Networks, Telecom Italia Wireline Services, Telefonica Data Argentina S.A., Teleglobe Canada ULC, Telstra Internet, Tiscali UK Limited, Ultel LLC, Verizon Internet Services, Virgin Media, Virtual Communications Corporation, WINDSTREAM COMMUNICATIONS, Wind Telecomunicazioni spa

Bibliography

[AAL$^+$05] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Dns security introduction and requirements. Technical report, RFC 4033, March, 2005.

[ACNP07] Mark Allman, Ken Christensen, Bruce Nordman, and Vern Paxson. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *ACM HotNets*, 2007.

[AJCZ12] Vijay Kumar Adhikari, Sourabh Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *INFOCOM*, pages 2521–2525, 2012.

[AKP$^+$07] Mark Allman, Christian Kreibich, Vern Paxson, Robin Sommer, and Nicholas Weaver. The Strengths of Weaker Identities: Opportunistic Personas. In *Proc. of USENIX Workshop on Hot Topics in Security (HotSec)*, August 2007.

[AL01] Paul Albitz and Cricket Liu. DNS and BIND, Fourth Edition. `http://docstore.mik.ua/orelly/networking_2ndEd/dns/ch10_07.htm`, April 2001.

[Ale] Alexa. `http://www.alexa.com/topsites`.

[All07] Mark Allman. Personal Namespaces. In *ACM SIGCOMM HotNets*, November 2007.

[AMSU10] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Comparing DNS Resolvers in the Wild. In *Proceedings of the 10th annual conference on Internet measurement*, pages 15–21. ACM, 2010.

[AMSU11] Bernhard Ager, Wolfgang Mühlbauer, Georgios Smaragdakis, and Steve Uhlig. Web content cartography. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet measurement*, pages 585–600, 2011.

[AN02] Jari Arkko and Pekka Nikander. Weak Authentication: How to Authenticate Unknown Principals without Trusted Parties. In *Security Protocols Workshop*, April 2002.

[And98a] M. Andrews. Negative caching of DNS queries (DNS NCACHE). 1998.

[And98b] M. Andrews. Negative Caching of DNS Queries (DNS NCACHE), March 1998. RFC 2308.

[APJ05] Sharad Agarwal, Venkata Padmanabhan, and Dilip A. Joseph. SureMail: Notification Overlay for Email Reliability. In *ACM HotNets*, November 2005.

[Bam] The Bamboo distributed hash table. http://bamboo-dht.org/.

[BCN01] N. Brownlee, KC Claffy, and E. Nemeth. Dns measurements at a root server. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, volume 3, pages 1672–1676. IEEE, 2001.

[Bel95] S.M. Bellovin. Using the domain name system for system break-ins. In *Proceedings of the Fifth Usenix UNIX Security Syposium, Salt Lake City, UT*, 1995.

[Ber] Dan Bernstein. `http://cr.yp.to/djbdns/notes.html`.

[BFV10] S. Burnett, N. Feamster, and S. Vempala. Chipping Away at Censorship Firewalls With User-Generated Content. In *Proc. 19th USENIX Security Symposium, Washington, DC*, 2010.

[BLCL$^+$94] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, 1994.

[BPSM$^+$00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1. 0 , w3c. recommendation 2000. *The Role of Citizen Cards in e-Government*, 455, 2000.

[CAR12] T. Callahan, M. Allman, and M. Rabinovich. Pssst, over here: Communicating without fixed infrastructure. In *INFOCOM, 2012 Proceedings IEEE*, pages 2841 –2845, march 2012.

[CARB11] Tom Callahan, Mark Allman, Michael Rabinovich, and Owen Bell. On grappling with meta-information in the internet. *SIGCOMM Comput. Commun. Rev.*, 41(5):13–23, October 2011.

[CCR$^+$03] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[ccz] Case Connection Zone. `http://caseconnectionzone.org/`.

[cho] Chord. http://pdos.csail.mit.edu/chord/.

[chr] DNS Prefetching - The Chromium Projects. `http://www.chromium.org/developers/design-documents/dns-prefetching`.

[cli] GNU CLISP - an ANSI Common Lisp Implementation. http://www.clisp.org/.

[CWFC08] S. Castro, D. Wessels, M. Fomenkov, and K. Claffy. A day at the root of the internet. *ACM SIGCOMM Computer Communication Review*, 38(5):41–46, 2008.

[DAV$^+$08] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS forgery resistance through 0x20-bit encoding: security via leet queries. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 211–222. ACM, 2008.

[DG08] C.G. Dickey and C. Grothoff. Bootstrapping of Peer-to-Peer Networks. In *Applications and the Internet, 2008. SAINT 2008. International Symposium on*, pages 205–208. IEEE, 2008.

[DMP$^+$02] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally Distributed Content Delivery. *Internet Computing, IEEE*, 6(5):50–58, 2002.

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proc. USENIX Security Symposium*, 2004.

[DOK92] P.B. Danzig, K. Obraczka, and A. Kumar. An analysis of wide-area name server traffic: a study of the internet domain name system. In *ACM SIGCOMM Computer Communication Review*, volume 22, pages 281–292. ACM, 1992.

[DPLL08a] D. Dagon, N. Provos, C.P. Lee, and W. Lee. Corrupted DNS resolution paths: The rise of a malicious resolution authority. In *Proceedings of Network and Distributed Security Symposium (NDSS'08)*, 2008.

[DPLL08b] D. Dagon, N. Provos, C.P. Lee, and W. Lee. Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority. In *Proc. of Network and Distributed Security Symposium*, 2008.

[DW09] J. Dinger and O. Waldhorst. Decentralized Bootstrapping of P2P Systems: A Practical View. *NETWORKING 2009*, pages 703–715, 2009.

[E+99] D.E. Eastlake et al. Rfc 2535: Domain name system security extensions. 1999.

[edu] Educause. `http://www.educause.edu/`.

[EK97] D. Eastlake and C. Kaufman. Rfc 2065: Domain name system security extensions. Technical report, RFC, IETF, Januar 1997, ftp://ftp. isi. edu/in-notes/rfc2065. txt, 1997.

[FS] F-Secure. Trojan: W32/dnschanger. `http://www.f-secure.com/v-descs/dnschang.shtml`.

[FSLL+06] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[GC11] O. Gudmundsson and SD Crocker. Observing dnssec validation in the wild. *Securing and Trusting Internet Names (SATIN)*, 2011.

[GNU] GNU. C library: Host names. `http://www.gnu.org/software/libc/manual/html_node/Host-Names.html`.

[gooa] Google Public DNS. `https://developers.google.com/speed/public-dns/`.

[Goob] Inc. Google. `https://developers.google.com/speed/public-dns/docs/performance#loadbalance`.

[Gooc] Inc. Google. `https://developers.google.com/speed/public-dns/docs/security`.

[HMLG11] Cheng Huang, D.A. Maltz, Jin Li, and A. Greenberg. Public DNS system and global traffic management. In *IEEE INFOCOM*, pages 2615 –2623, 2011.

[HSF85] K. Harrenstien, M. Stahl, and E. Feinler. Dod internet host table specification, 1985.

[HV04] T. Hansen and G. Vaudreuil. Message Disposition Notification, May 2004. RFC 3798.

[JBB+09] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. *ACM Trans. on the Web*, 3(1), 2009.

[JSBM02] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. *Networking, IEEE/ACM Transactions on*, 10(5):589–603, 2002.

[JST+09] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, and R.L. Braynard. Networking Named Content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[Kam04] Dan Kaminsky. Black Ops of DNS. Black Hat Briefings, 2004.

[Kam08a] D. Kaminsky. Black ops 2008: It's the end of the cache as we know it. *Black Hat USA*, 2008.

[Kam08b] Dan Kaminsky. Black Ops 2008: It's the end of the Cache as we know it. Black Hat Briefings, 2008.

[KCC+07] T. Koponen, M. Chawla, B.G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (And Beyond) Network Architecture. *ACM SIGCOMM*, 2007.

[KGPP10] A. Kalafut, M. Gupta, P.Rattadilok, and P. Patel. Surveying Wildcard Usage Among the Good, the Bad, and the Ugly. In *SecureComm*, 2010.

[Kud74] MD Kudlick. Host names on-line. 1974.

[KWGA09] Joakim Koskela, Nicholas Weaver, Andrei Gurtov, and Mark Allman. Securing Web Content. In *ACM CoNext Workshop on ReArchitecting the Internet (ReArch)*, December 2009.

[Lam73] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lev66] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LHF$^+$07] Z. Liu, B. Huffaker, M. Fomenkov, N. Brownlee, and K. Claffy. Two days in the life of the dns anycast root servers. *Passive and Active Network Measurement*, pages 125–134, 2007.

[lib] Liberty alliance project. http://www.projectliberty.org/.

[LL10a] D. Leonard and D. Loguinov. Demystifying service discovery: implementing an Internet-wide scanner. In *Proceedings of the 10th ACM Conference on Internet Measurement*, pages 109–122. ACM, 2010.

[LL10b] D. Leonard and D. Loguinov. Demystifying Service Discovery: Implementing an Internet-Wide Scanner. In *Proceedings of the 10th annual conference on Internet measurement*, pages 109–122. ACM, 2010.

[LSZ02] R. Liston, S. Srinivasan, and E. Zegura. Diversity in dns performance measures. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 19–31. ACM, 2002.

[LWM07] Graham Lowe, Patrick Winters, and Michael L Marcus. The great dns wall of china. *MS, New York University. Accessed December*, 21, 2007.

[Maxa] MaxMind. City-Level GeoIP Database. `http://dev.maxmind.com/geoip/geolite`.

[Maxb] MaxMind. GeoIP ISP Database. `http://www.maxmind.com/en/isp`.

[MD88] P. Mockapetris and K.J. Dunlap. *Development of the domain name system*, volume 18. ACM, 1988.

[Moc83a] P. Mockapetris. Rfc 882: Domain names: Concepts and facilities. 1983.

[Moc83b] P. Mockapetris. Rfc 883: Domain names: Implementation specification. 1983.

[Moc87a] P. Mockapetris. Domain names implementation and specification. rfc 1035. *http://www.ietf.org/rfc/rfc1035.txt*, 1987.

[Moc87b] Paul Mockapetris. Domain Names - Concepts and Facilities, November 1987. RFC 1034.

[Moc87c] Paul Mockapetris. Domain Names - Implementation and Specification, November 1987. RFC 1035.

[Moo03] K. Moore. Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs), January 2003. RFC 3461.

[opea] OpenID Attribute Exchange. `http://openid.net/specs/openid-attribute-exchange-1_0.html`.

[Opeb] OpenDNS - A Technical Overview. `http://www.opendns.com/technology`.

[Ora] Oracle. Java api: Inetaddress.getbyname(). `http://docs.oracle.com/javase/6/docs/api/java/net/InetAddress.html#getByName(java.lang.String)`.

[PAS+04a] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the responsiveness of DNS-based network control. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 21–26, 2004.

[PAS+04b] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the Responsiveness of DNS-based Network Control. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, December 1999.

[PS10] Vern Paxson and Robin Sommer. Personal Communication, July 2010.

[PUK+11] Ingmar Poese, Steve Uhlig, Mohamed Ali Kaafar, Benoit Donnet, and Bamba Gueye. Ip geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[RF08] A. Ramachandran and N. Feamster. Authenticated Out-of-Band Communication over Social Links. In *ACM SIGCOMM Workshop on Online Social Networks*, 2008.

[RFR90] W. Richard, Bill. Fenner, and A.M. Rudoff. *UNIX network programming*. Addison-Wesley, 1990.

[RL96] R.L. Rivest and B. Lampson. Sdsi-a simple distributed security infrastructure. Crypto, 1996.

[RMTP08] Moheeb Rajab, Fabian Monrose, Andreas Terzis, and Niels Provos. Peeking through the cloud: Dns-based estimation and its applications. In *Applied Cryptography and Network Security*, pages 21–38. Springer, 2008.

[SAZ$^+$04] Ion Stoica, Daniel Adkins, Shelley Zhaung, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *IEEE/ACM Transactions on Networking*, volume 12, April 2004.

[Sch93] C. Schuba. *Addressing weaknesses in the domain name system protocol*. PhD thesis, Purdue University, 1993.

[SCKB06] A.J. Su, D.R. Choffnes, A. Kuzmanovic, and F.E. Bustamante. Drafting Behind Akamai (Travelocity-based Detouring). *COMPUTER COMMUNICATION REVIEW*, 36(4):435, 2006.

[SHLX03] Arnaud Sahuguet, Richard Hull, Daniel F. Lieuwen, and Ming Xiong. Enter once, share everywhere: User profile management in converged networks. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[SKAT12a] C.A. Shue, A.J. Kalafut, M. Allman, and C.R. Taylor. On building inexpensive network capabilities. *ACM SIGCOMM Computer Communication Review*, 42(2):72–79, 2012.

[SKAT12b] Craig Shue, Andrew Kalafut, Mark Allman, and Curtis Taylor. On Building Inexpensive Network Capabilities. *ACM SIGCOMM Computer Communication Review*, 42(2), April 2012.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. USENIX Security Symposium*, 2002.

[SS10] S. Son and V. Shmatikov. The hitchhikerŠs guide to dns cache poisoning. *Security and Privacy in Communication Networks*, pages 466–483, 2010.

[SSDA12] Matt Sargent, Brian Stack, Tom Dooner, and Mark Allman. A First Look at 1 Gbps Fiber-To-The-Home Traffic. Technical Report 12-009, International Computer Science Institute, August 2012.

[SSV06] S. Singh, S. Shenker, and G. Varghese. Service portability: Why http redirect is the model for the future. In *ACM SIGCOMM HotNets*, 2006.

[Sta12] Brian Stack. Personal Communication, May 2012.

[SZ11] Yuval Shavitt and Noa Zilberman. A geolocation databases study. *Selected Areas in Communications, IEEE Journal on*, 29(10):2044–2056, 2011.

[TAQR09] S. Triukose, Z. Al-Qudah, and M. Rabinovich. Content Delivery Networks: Protection or Threat? In *ESORICS*, pages 371–389, 2009.

[TCL08] Dinh Nguyen Tran, Frank Chiang, and Jinyang Li. Friendstore: cooperative online backup using trusted nodes. In *Proc. of the 1st Workshop on Social Network Systems*, 2008.

[TSGW09] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: better privacy for social networks. In *CoNEXT*, 2009.

[TWR11] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. In *Proceedings of the 20th International Conference on World Wide Web*, pages 467–476, 2011.

[ucn] ICSI/Case User-Centric Networking Project. `http://www.icir.org/user-centric-networking/`.

[Wea12] Nicholas Weaver. personal communication, 2012.

[WF03] D. Wessels and M. Fomenkov. Wow, thatŠsa lot of packets. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, 2003.

[win] Windows Registry. `http://technet.microsoft.com/en-us/library/cc751049.aspx`.

[WKNP11] Nicholas Weaver, Christian Kreibich, Boris Nechaev, and Vern Paxson. Implications of Netalyzr's DNS Measurements. In *First Workshop on Securing and Trusting Internet Names (SATIN)*, NPL, Teddington, UK, April 2011.

[WKP11] Nicholas Weaver, Christian Kreibich, and Vern Paxson. Redirecting DNS for Ads and Profit. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, San Francisco, CA, USA, August 2011.

[WMS03] C.E. Wills, M. Mikhailov, and H. Shang. Inferring relative popularity of internet applications by actively querying dns caches. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 78–90. ACM, 2003.

[WS00] C. Wills and H. Shang. The contribution of dns lookup costs to web object retrieval. Technical report, Citeseer, 2000.

[WSJBF10] D.I. Wolinsky, P. St Juste, P.O. Boykin, and R. Figueiredo. Addressing the P2P Bootstrap Problem for Small Overlay Networks. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2010.

[ZHR$^+$11] C. Zhang, C. Huang, K. Ross, D. Maltz, and J. Li. Inflight modifications of content: who are the culprits? In *Workshop of Large-Scale Exploits and Emerging Threats (LEET'11)*, 2011.

[Zyt] Zytrax.com. Dns bind operations statements. `http://www.zytrax.com/books/dns/ch7/hkpng.html`.