Psst, Over Here: Communicating Without Fixed Infrastructure

Tom Callahan,[†] Mark Allman,[*] and Michael Rabinovich[†]

TR-12-002

January 2012

Abstract

This paper discusses a way to communicate 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 (e.g., for censorship purposes). 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. We conduct several experiments that illustrate the efficacy of our techniques to exchange an IP address that is presumed to be a rendezvous point for future communication. Additionally, we describe a wider channel that can be used to transmit an SMS- or Twitter-like 140-character message.

† Case Western Reserve University, Cleveland, OH 44106
* ICSI, Berkeley, CA 94704

This work was partially supported by funding provided through National Science Foundation grants CNS: 0433702 ("Center for Internet Epidemiology and Defenses"); CNS: 0831821 ("Relationship-Oriented Networking"); CNS: 0831535 ("Comprehensive Applications Analysis and Control"); and CNS: 0831780 ("Relationship-Oriented Networking").. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

I. 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 [1], [2] 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 paper 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 easy-to-find and highly robust 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 trackerless 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 paper 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 § II). 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 paper we show we can accurately publish and query for such information.

II. 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. 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:

$$G(x) = sha1(sha1(S) + x), \tag{1}$$

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 hosts—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].

In our scanning experiment we probed from roughly 80 PlanetLab 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 99^{th} percentile being 1,284. These results make probing tractable for our purposes because even scanning at a low rate will turn up multiple servers. E.g., sending one probe per minute over the course of one day will yield five DNS servers on average. Further, once the set of servers is obtained, it can be maintained with even lower-rate probes over longer time scale. In addition, DNS servers that simply disappear (as has been noted elsewhere in the literature [3], [4]) will be readily detected as attempts are made to store information at the given server. Such knowledge can also be used to trigger a new server detection phase.

While in this paper 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 allow us to find a large number of usable DNS servers quite rapidly. For instance, we conducted a small experiment that was 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 [5]). 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 § I of not requiring a central hub to bootstrap communication. However, we note that the approach can first be viewed as an optimization and 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).

III. 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 § II to find suitable recursive DNS servers and, as they are found, publish and retrieve 32-bit messages.

A. 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 1. We also need a domain we know to

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

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

support wildcard DNS queries, that is, queries for unknown 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 [6], 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 § V.

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:
$$R_i = G(\text{"}Record\%d\text{"}, i), \tag{2}$$
where "Record" is just an arbitrary identifier that all actors involved know (here and in the rest of the paper we use a `printf`-like notation to compose strings).
2) Similarly, we generate a "barrier record" using:
$$B = G(\text{"}Barrier''\text{)} \tag{3}$$
3) Next we form sets of bit numbers, 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 [7]. 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:
$$R_i = G(\text{"}Record\%d\text{"}, i), \tag{4}$$
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	12.9K
Non-Responsive Servers	742	520
Non-Recursive Servers	2.6K	1.7K
Non-Decrementing TTL	15K	9.8K
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 I
BIT-PIPE PUBLICATION RESULTS

B. Results

We tested 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 § II 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.

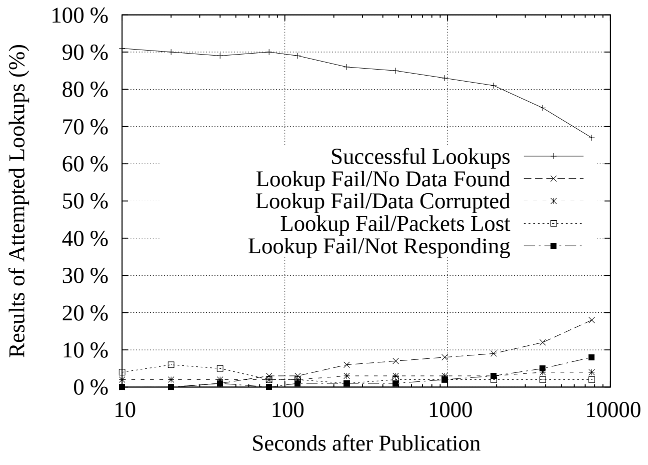
After each publication the host waits 10 seconds and then retrieves the message to assess the efficacy of the data insertion process. Table I shows the results of our publication attempts. First we note that in spite of our efforts (sketched in § II) to identify unusable DNS servers during the scanning phase we still ended up with problems in roughly 15% of the servers we tried. The largest problems come from TTL decrementing issues—even though we tried to weed out servers with this issue during our scanning pass (§ II)[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 the efficacy of our information sharing technique, they do illustrate that one must exercise some 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 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.

Figure 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. The remainder of the failure causes remain fairly constant and relatively small across the time period.

[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.



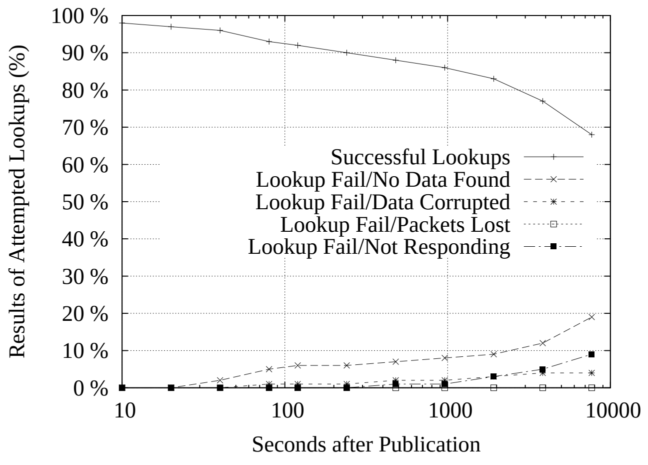


Fig. 1. Bit pipe retrieval results for the TTL (top) and Recursion Desired methods.

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.

C. Discussion

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

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 the corruption rate is less than half the message across both our methods at both mean and median. 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 § IV. 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.

IV. 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 § I). 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 § I—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 accounts for 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 § III we rely on a secret S, a generator function $G(x)$ (defined in equation 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).

A. 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:
$$B_i = G(''TwitterBarrier\%d'', i) \tag{5}$$
2) We also generate names for each character of the message (1–140) as above:
$$R_j = G(''TwitterQuery\%d'', j) \tag{6}$$
3) Finally, we iterate through the symbol values from $k = 1 \ldots 56$. 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

[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 and could be ignored.

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:

$$R_i = G(\text{"}TwitterQuery\%d\text{"}, i) \tag{7}$$

2) Similarly, we generate a "base record" using:

$$B = G(\text{"}TwitterBase\text{"}) \tag{8}$$

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 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.

B. Results

We coupled our Twitter-like message exchanges with the DNS scanning outlined in § II. 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 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.

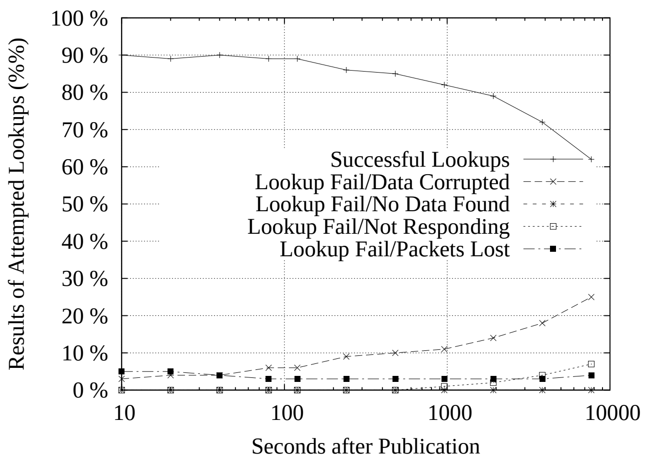
	Explicit	**Implicit**
Pub. Attempts	80K	86K
Unusable Servers	10K	13K
Non-Responsive Servers	461	298
Non-Recursive Servers	981	5
Non-Decrementing TTL	8.5K	11K
Weird TTL Decrementing	816	1.4K
Usable Servers	70K	73K
Successful	53K	47K
Failure: Packet Loss	6.6K	6.3K
Failure: No Data Found	1	1
Failure: Corrupt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

TABLE II
TWITTER-LIKE PUBLICATION RESULTS

Table II 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 ended up 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 II). 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 [8] 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 other PlanetLab nodes to retrieve the messages at various times from various places just as we did in § III. The results for the explicit and implicit methods



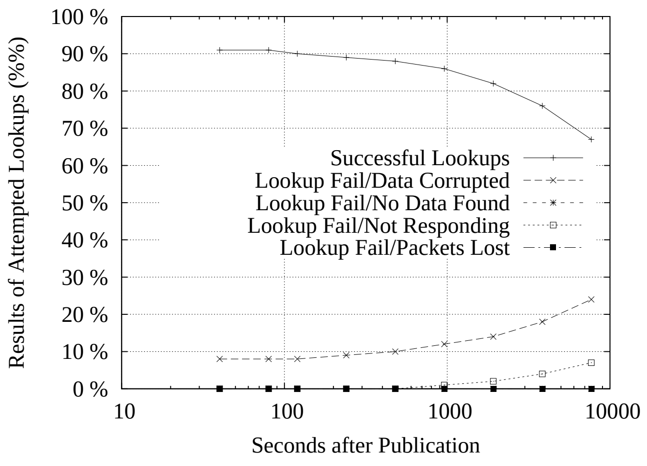


Fig. 2. Explicit (top) and implicit retrieval results.

are shown in Figure 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 the cached entries age. 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 looked at the retrieved messages and determined that if the message experienced some corruption but the Levenshtein edit distance [8] 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 of messages that are nearly successful is roughly constant over time at 2% and 6% for the explicit and implicit schemes, respectively. Finally, we note that 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 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. (Note, we are planning another set of experiments that will more verbosely record corruption and loss information such that we will have a proper treatment rather than relying on bounds in the final version of this paper.)

As described in § III 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

[6]Note, due to a measurement glitch the first two time intervals for the implicit mechanism are not available. While we expect these results will be highly similar to those reported for the 40 second interval (first point on the x-axis on the lower plot), we will endeavor to re-run the experiments to include these two points.

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

which also aligns with our PlanetLab measurements. This double-check suggests the PlanetLab platform is not overtly impacting our results.

V. Additional Considerations

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

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. Even if an eavesdropper intercepts the message it will be meaningless without the secret used by $G()$.

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 § III-B, we can replace random scanning with hit list scanning, which both produces less volume and more valid DNS traffic (i.e., little traffic to hosts that to 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 paper: (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. However, where the publication process is executed is somewhat easier to control since it is only needed at one location. Further, even without scanning the publisher will be required to refresh the data in the system periodically which requires a large number of queries occurring in a fairly tightly defined time window which will be quite 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.

Other Channels: An additional avenue for encoding information is to implicitly leverage DNS caching (both of valid records and invalid records [9]). 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 III 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 [10]). 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), however, timing issues could complicate the implementation. We leave a full treatment for future work.

VI. Related Work

As noted in § III the only directly related work we are aware of is that given in a Black Hat presentation [7]. One slide of the talk describes the RD method sketched in § III. We are not aware of any empirical evaluation of the idea on that slide and therefore we conduct such an evaluation in this paper.

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 [11] is an innovative 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 [12], we note several similarities in emphasis between the two systems. For example, competent network operators will be able to prevent the usage of both Tor and our system. (Despite this roadblock, we note that Tor has been able to attain a large user base, and 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, it will be hard for an external monitor to ascertain the identity of an actor in our scheme as well (the DNS server used will often be in a different jurisdiction, lack the necessary logs, etc).

With respect to the bootstrapping application of our scheme, we note some recent work in this area. For example, [13] 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 [14], 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 recursive DNS servers and use these to bootstrap, rather than scanning for the (possibly small) number of peer nodes directly. In [15], the authors propose using well-known P2P networks for bootstrapping smaller networks, and perform evaluations of several of these services. Our scheme rides on a more essential part of the Internet infrastructure (and in fact, as we outlined earlier, can be extended to use other parts of the Internet infrastructure such as CDNs).

VII. Conclusions

This paper 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. Additionally, we require domains that offer wildcard DNS entries—which again are widespread. 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 paper 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.

VIII. Acknowledgments

This work benefits from discussions with Vern Paxson and Nicholas Weaver and the comments from the anonymous reviewers. This work was supported in part by NSF grants CNS-0433702, CNS-0831821, CNS-0831535 and CNS-0831780.

References

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

[2] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking Named Content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.

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

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

[5] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the Internet in Your Spare Time," in *Proc. USENIX Security Symposium*, 2002.

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

[7] D. Kaminsky, "Black Ops of DNS," Black Hat Briefings, 2004.

[8] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[9] M. Andrews, "Negative Caching of DNS Queries (DNS NCACHE)," Mar. 1998, rFC 2308.

[10] S. Triukose, Z. Al-Qudah, and M. Rabinovich, "Content Delivery Networks: Protection or Threat?" in *ESORICS*, 2009, pp. 371–389.

[11] 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.

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

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

[14] C. Dickey and C. Grothoff, "Bootstrapping of Peer-to-Peer Networks," in *Applications and the Internet, 2008. SAINT 2008. International Symposium on*. IEEE, 2008, pp. 205–208.

[15] D. Wolinsky, P. St Juste, P. 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*. IEEE, 2010, pp. 1–10.