LDplayer: DNS Experimentation at Scale

USC/ISI Technical Report ISI-TR-722, Novemeber 2017 *

Liang Zhu John Heidemann
USC/Information Sciences Institute

Abstract

DNS has evolved over the last 20 years, improving in security and privacy and broadening the kinds of applications it supports. However, this evolution has been slowed by the large installed base with a wide range of implementations that are slow to change. Changes need to be carefully planned, and their impact is difficult to model due to DNS optimizations, caching, and distributed operation. We suggest that *experimentation at scale* is needed to evaluate changes and speed DNS evolution. This paper presents LDplayer, a configurable, general-purpose DNS testbed that enables DNS experiments to scale in several dimensions: many zones, multiple levels of DNS hierarchy, high query rates, and diverse query sources. LDplayer provides high fidelity experiments while meeting these requirements through its distributed DNS query replay system, methods to rebuild the relevant DNS hierarchy from traces, and efficient emulation of this hierarchy of limited hardware. We show that a single DNS server can correctly emulate multiple independent levels of the DNS hierarchy while providing correct responses as if they were independent. We validate that our system can replay a DNS root traffic with tiny error ($\pm 8 ms$ quartiles in query timing and $\pm 0.1\%$ difference in query rate). We show that our system can replay queries at 87k queries/s, more than twice of a normal DNS Root traffic rate, maxing out one CPU core used by our customized DNS traffic generator. LDplayer's trace replay has the unique ability to evaluate important design questions with confidence that we capture the interplay of caching, timeouts, and resource constraints. As an example, we can demonstrate the memory requirements of a DNS root server with all traffic running over TCP, and we identified performance discontinuities in latency as a function of client RTT.

1 Introduction

The Domain Name System (DNS) is critical to the Internet. It resolves human-readable names like www.iana.org to IP addresses like 192.0.32.8 and service discovery for many protocols. Almost all activity on the Internet, such as web-browsing and e-mail, depend on DNS for the correct operations. Beyond name-to-address mapping, DNS today has grown to play various of broader roles in the Internet. It provides query engine for anti-spam [14] and replica selection for content delivery networks (CDNs) [19]. DANE (DNS-based Authentication of Named Entities) [11] provides additional source of trust by leveraging the integrity verification of DNSSEC [3]. The wide use and critical role of DNS prompt its continuous evolution.

However, evolving the DNS protocol is challenging because it lives in a complex ecosystem of many implementations, archaic deployments, and interfering middleboxes. These challenges increasingly slow DNS development: for example, DNSSEC has taken a decade to deploy [17] and current use of DANE is growing but still small [25]. Improvements to DNS privacy are needed [5] and now available [24, 12], but how long will deployment take?

DNS performance issues are also a concern, both for choices about protocol changes, and for managing inevitable changes in use. There are a number of important open questions: How does current server operate under the stress of a Denial-of-Service (DoS) attack? What is the server and client performance when protocol or architecture changes? What if all DNS requests were made over QUIC, TCP or TLS? What about when DNSSEC keys change size?

*Research by Liang Zhu and John Heidemann in this paper is partially sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, HSARPA, Cyber Security Division, BAA 11-01-RIKA and Air Force Research Laboratory, Information Directorate under agreement number FA8750-12-2-0344, and contract number D08PC75599. The U.S. Government is authorized to make reprints for Governmental purposes notwithstanding any copyright.The views contained herein are those of the authors and do not necessarily represent those of DHS or the U.S. Government.

Ideally models would guide these questions, but DNS is extraordinarily difficult to model because of interactions of caching and implementation optimizations across levels of the DNS hierarchy and between clients and servers. Estimating system-wide costs are also quite difficult: for example, accounting for the memory of TCP and TLS is difficult since it is split across the kernel, libraries, and in the application.

Definitive answers to DNS performance therefore require end-to-end *controlled experiments* and trace replay. Experiments enable testing different approaches for DNS and evaluating the costs and benefits against different infrastructures, revealing unknown constraints. *Trace replay* can drive these experiments with real-world current workloads, or with extrapolated "what-if" workloads.

Accurate DNS experiments are quite challenging. In addition to the requirements of modeling, the DNS system is large, distributed, and optimized. With millions of authoritative and recursive servers, it is hard to recreate a global DNS hierarchy in a controlled experiment. A naive testbed would therefore require millions of separate servers, since protocol optimizations cause incorrect results when many zones are provided by one server. Prior DNS testbeds avoided these complexities, instead studying DNSSEC overhead in a piece of the tree [2] and query distribution of recursive servers [22]. While effective for their specific topics, these approaches do not generalize to support changing protocols, large query rates, and diverse query sources across a many-level hierarchy.

In this paper, we present *LDplayer*, a configurable, general-purpose DNS testbed that enables DNS experiments at scale in several dimensions: many zones, numerous levels of DNS hierarchy, large query rates, and diverse query sources. Our system provides a basis for DNS experimentation that can further lead to DNS evolution.

Our first contribution is to show how LDplayer can scale to efficiently model a large DNS hierarchy and playback large traces (§2). LDplayer can can correctly emulate multiple independent levels of the DNS hierarchy on a single instance of DNS server, exploiting a combination of proxies and routing to circumvent optimizations that would otherwise distort results. Our insight is that a single server hosting many different zones reduces deployment cost; we combine proxies and controlled routing "*pass*" queries to the correct zone so that the server gives the correct answers from a set of different zones. To this framework we add a two-level query replay system where a single computer can accurately replay more than 87 k queries per second, twice as fast as typical query rates a DNS root letter. Multiple computers can generate traffic at 10-100$\times$ that rate.

Second, the power of controlled replay of traces is that we can modify the replay to explore "what if" questions in DNS evolution (§5). We demonstrate this capability with two experiments. We explore how traffic volume changes if all DNS queries employ DNSSEC. We also use LDplayer to consider how server memory and client latency changes if all queries were TCP instead of UDP. Other potential applications include the study server hardware and software under denial-of-service attack, growth of the number or size of zones, or changes in hardware and software. All of these questions are important operational concerns today. While some have been answered through one-off studies and custom experiments or analysis, LDplayer allows evaluation of actual server software, providing greater confidence in the results. For example, relative to prior studies of DNS over TCP [24], our use of trace-replay provides strong statements about all aspects of server memory usage and discovers previously unknown discontinuities in client latency.

2 LDplayer: DNS trace player

We next describe our requirements, then summarize the and architecture and describe critical elements in detail.

2.1 Design Requirements

The goal of LDplayer is to provide a controlled testbed for repeatable experiments upon realistic evaluation of DNS performance, with the following requirements:

Emulate complete DNS hierarchy, efficiently: LDplayer must emulate multiple independent levels of the DNS hierarchy and provide correct responses using minimal commodity hardware.

We must support many zones. It is not scalable to use separated servers or virtual machines to host each zone because of hardware limits and many different zones in a network trace. A single server providing many zones of DNS hierarchy does not work directly, because the server gives the final DNS answer straightly and skips the round trip of DNS referral replies.

Replays do not leak traffic to the Internet: Experimental traffic must stay inside the testbed, without polluting the Internet. Otherwise each experiment could leak bursts of requests to the real Internet, causing problems for the Internet and the experiment. For the Internet, leaks of replay from high-rate experiments might stress real-world servers. For the experiment, we need to control response times, and queries that go to the Internet add uncontrolled delay and jitter.

Repeatability of experiments: LDplayer needs to support repeatable, controlled experiments. When an experiment is re-run, the replies to the same set of query replay should stay the same. This reproducibility is very important for experiments that require fixed query-response content to evaluate new transform in DNS, such as protocol changes and new server implementations. Without building complete zone, the responses could change over time when re-looked up. Some zones hosted at CDNs may have external factors that influence responses, such as load balancing.

Controlled variations in traffic, when desired: Replay must be able to manipulate traces to answer "what if" questions with variations of real traffic. Since input is normally binary network trace files, the main challenge is how to provide a flexible and user-friendly mechanism for query modification. We also need to minimize the delay by query manipulation, so that input processing is fast enough to keep up with real time.

Accurate timing at high query rates: LDplayer must be capable of replaying queries at fast rates, while preserving correct timing, to reproduce interesting real-world traffic patterns for both regular and under attack. However, both using a single host and many hosts have challenges. Due to resource constraints on CPU and the number of ports, a single host may not be capable to replay fast query stream or emulate diverse sources. A potential solution is to distribute input to different hosts, however, it brings another challenge in ensuring the correct timing and ordering of individual queries.

Support multiple protocols effectively: LDplayer needs to support both connectionless (UDP) and connection-oriented (TCP and TLS), given increasing interest in DNS over connections [24]. However, connection-oriented protocols bring challenges in trace replay: emulating connection reuse and round-trip time (RTT). The query replay system of LDplayer is the first system that can emulate connection reuse for DNS over TCP. Emulation of RTT is important for experiments of connection-oriented DNS, because RTT will affect protocol responses with extra messages for connection setup, while connectionless protocols do not incur those extra messages.

2.2 Architecture Overview

We next describe LDplayer's architecture (Figure 1). With captured network traces of DNS queries (required) and responses (optional), a researcher can use our *Zone Constructor* to generate required zone files. LDplayer uses a logically *single* authoritative DNS server with proxies to emulate entire DNS hierarchy (*Hierarchy Emulation*). The single DNS server provides all the generated zone files. The proxies manipulate packet addresses to achieve successful interaction between the recursive and authoritative servers, such as providing correct answers to replayed queries. As a distributed query system, the *Query Engine* replays queries in the captured traces. Optionally, the researcher can use *Query Mutator* to change the original queries arbitrarily for different replay, and query mutator can run live with query replay.

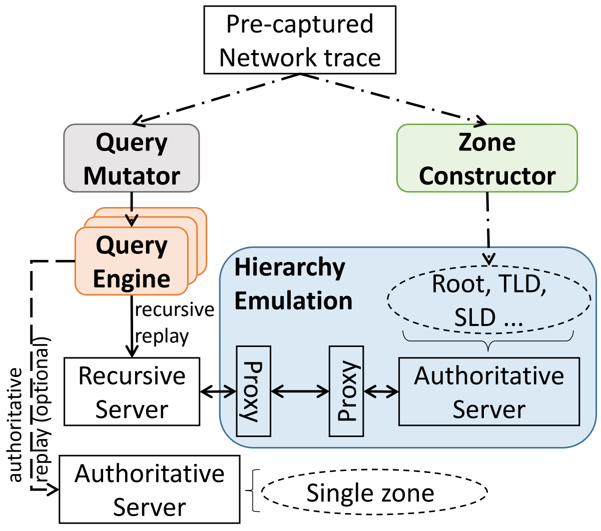


Figure 1: LDplayer architecture

Each component in LDplayer addresses a specific design requirement from §2.1. In LDplayer's zone constructor, we *synthesize data for responses* and generate required zone files by performing one-time fetch of missing records (§2.3). We run a real DNS server that hosts these reusable zone files and provides answers to replayed queries, so that we can get repeatable experiments without disturbing the Internet.

With generated zone files, we need to *emulate DNS hierarchy* to provide correct answers. Logically, we want many server hosts, one per each zone, like the real world. However, we compress those down to a single server process with single network interface using split-horizon DNS [1], so that the system scales to many zones. For easy deployment, we redirect the replayed experimental traffic to proxies, which then manipulate packet addresses to simplify routing configuration and discriminate queries for different zones to get correct responses (§2.4). We could run multiple instances of the server to support large query rate and massive zones, with routing configuration that redirects queries to the correct servers.

In LDplayer's query mutator, we pre-process the trace so that query manipulation does not limit replay times. We convert network traces to human-readable plain text for flexible and user-friendly manipulation. After necessary query changes, we convert the result text file to a customized binary stream of internal messages for fast query replay (§2.5). In principle, at lower query rates, we could manipulate a live query stream in near real time.

In LDplayer's query engine, we use a central controller to coordinate queries from many hosts and synchronize the time between the end queriers, so that LDplayer can replay large query rates accurately. The query engine can replay queries via different protocols (TCP or UDP) effectively. We distribute queries from the same sources in the original trace to the same end queriers for replay, in order to emulate queries from the same sources which is critical for connection reuse (§2.6). LDplayer replays queries based on the timing in the original trace without preserving query dependencies.

2.3 Synthesize Zones to Provide Responses

To support experiment repeatability and avoid leaking bulk experimental DNS queries to the Internet, we build the zone files that drive the experiment *once* and then reuse them in each experiment. We build zones by replaying the queries, once, against the real-world servers on the Internet and harvesting these responses.

One-time Queries to the Internet: We need to build a DNS hierarchy that includes answers to all the queries that will be made during replay. When emulating an authoritative server, we can often acquire the zone from its manager, but when emulating recursive servers we must recreate all zones that will be queried. (If any part of hierarchy is missing, replayed queries may fail.) For example, if `.com` delegation (`NS` records of `.com`) is missing in the root zone, a recursive server will fail to answer all the queries for `.com` names in experiments.

To build a DNS hierarchy that covers all queries, we send all unique queries in the original trace to a recursive server with cold cache and allow it to query Internet to satisfy each query. We then capture the DNS responses all authoritative servers that respond, recording traffic at the upstream network interface of the recursive server. Since the recursive server walks down the DNS hierarchy for each queries, the captured trace contains all authoritative data needed to build zones for the parts of the DNS hierarchy that are needed for the replay.

Zone construction need to be done only once (we save the recreated zones for reuse) so any load it places on the original servers is a one time cost. (We also prototyped an alternative that primes these zones with replies from the trace, but we found that caching makes raw traces incomplete. We therefore rebuild the entire zone from scratch to provide a consistent snapshot.)

Construct Zones from Traces: Given the traces captured at the recursive server, we next reverse the traces to recreate appropriate zone data.

We convert traces to multiple zone files, since a full DNS query (for example, `mail.google.com`) may touch several different servers (root, `.com`, `googlemail.l.google.com`, plus their authoritative nameservers, DNSSEC records, etc.).

We first scan the whole trace and identify authoritative nameservers (`NS` records) for different domains and their host addresses (`A` or `AAAA` records) from all the responses. Since most of domains have multiple nameservers (for example, `google.com` has 4 nameservers: `ns{1-4}.google.com`), a recursive server may choose any of them to trace the query based on its own strategy. We group the set of nameservers responsible for the same domain, and aggregate all DNS response data from the same group of nameservers by checking the source address in responses. We then generate an intermediate zone file from the aggregate data.

Since a nameserver can serve multiple different zones, the intermediate zone file we generate may contain data of different domains and may not be a valid zone file acceptable by a DNS server. We further split the response data in the intermediate zone file by different domains, and output corresponding separated zone files. Optionally we can also merge the intermediate zone files of multiple traces. To determine zone cuts (which parts of the hierarchy are served by different nameservers), we probe for `NS` records at each change of hierarchy.

Similarly, we can recreate a zone file for queries replaying at an authoritative server. Since only single authoritative server is involved without the recursive, the zone file reconstruction is straightforward.

Recover Missing Data: Sometimes records needed for a complete, valid zone will not appear in the traces. For example, a valid zone file needs `SOA` (Start of Authority) record and `NS` records for the zone, however, those records are not required for regular DNS use. We create a fake but valid `SOA` record and explicitly fetch `NS` records if they are missing.

Handle inconsistent replies: DNS queries sometimes vary over time, such as replies to CDNs that balance load across a cluster, or in the unlikely event that the zone is modified during our rebuild. DNS records can be updated. However sometimes those update conflict with each other, such as multiple `CNAME` records for the same name while only one allowed in principal. More often, the address mapping for names may change over time, such as content delivery network (CDN) redirecting by updating DNS using its own algorithm.

By default, To build a consistent zone, we choose the first answer when there are multiple differing responses. Simulating the various CDN algorithms to give different addresses for queries is future work.

2.4 Emulate DNS Hierarchy Efficiently

With zones created from traces, we next introduce how we emulate DNS hierarchy in order to answer replayed queries correctly in LDplayer. Handling queries to a recursive server requires emulating multiple hierarchical zones, while handling queries to an authoritative server does not need to emulate hierarchy due to a single zone.

The greatest challenges of emulating full DNS hierarchy in a testbed environment are scalability to support many different zones and easy deployment. Since we use

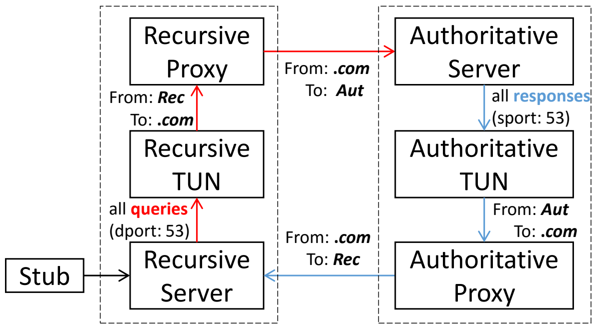


Figure 2: Server proxies manipulate the source and destination addresses in the queries and responses to make routing work and get the correct responses.

real DNS records (such as real public IP addresses) in zone files, the other challenge is how to make these zone files work in a private testbed environment with local IP addresses. A naive way would use separate authoritative servers for each zone, each on its own server. Even with virtual machines, such an approach cannot emulate typical recursive workloads that see hundreds or thousands of zones over days or weeks—it will encounter limits of memory and virtual network interfaces. We see 549 valid zones in a 1-hour trace Rec-17 (Table 1) captured at a department-level recursive server. DNS server software can host multiple zones in one server, but optimizations built into common server software mean that putting the whole hierarchy in one server gives different results. (Asking for `www.example.com` will directly produce an IP address from a server that stores the root, `.com`, and `example.com` zones, not three queries.)

Scale to many zones with a single server: To emulate complete DNS hierarchy efficiently, instead we contribute a *meta-DNS-server*: a *single authoritative server instance* with a *single network interface* correctly emulates multiple independent levels of DNS hierarchy using real zone files, while providing correct responses as if they were independent.

Challenges: There are some challenges in making the recursive server successfully interact with the meta-DNS-server during query replay, because we use a single server instance and a single network interface to provide authoritative service to *all* relevant zones in the trace.

First, how do the queries sent by the recursive server merge to the same network interface at meta-DNS-server? Typically, if a recursive receives an incoming query (for example, `www.google.com A`) with cold cache, it walks down the DNS hierarchy (for example, `root` $\rightarrow$ `com` $\rightarrow$ `google.com`) and sends queries to respective authoritative servers (for example, `a.root-servers.net` $\rightarrow$ `a.gtld-servers.net` $\rightarrow$ `ns1.google.com`). As a result, the queries out of the recursive have a set of different destination IP addresses. Without changes, those queries will not be routed to the meta-DNS-server by default.

Second, how does the meta-DNS-server know which zone files to use in order to answer the incoming queries correctly? When a recursive server resolves a incoming query iteratively with cold cache, the query content sent by the recursive is the same, regardless of which level of the DNS hierarchy it is contacting. Assume the meta-DNS-server receives a query (for example, `www.google.com A`) which was meant to send to the authoritative server of `com`. The meta-DNS-server is not able to identify the target zone (`com`) based on the query content. The answers from `root`, `com` and `google.com` zones are are completely different (a referral answer of `com`, a referral answer of `google.com`, and an authoritative answer of `www.google.com A` respectively). A wrong answer which is not from the correct zone (`com`) can lead to a broken hierarchy at the recursive and further failure of query replay.

Third, how are meta-DNS-server's responses accepted by the recursive server? Assume the meta-DNS-server can pick the correct zone (for example, `com`) to answer queries (we will present the solution later). All the reply packets by meta-DNS-server have the same meta-DNS-server's address as source IP addresses. Even if the recursive receives this "correct" reply, it will not accept the reply because the reply source address (the address of meta-DNS-server) is not matched with the original query destination address (for example, the address of `a.gtld-servers.net`)

Solutions: To overcome those challenges, at high level, we use split-horizon DNS [1] to host different zones discriminated by incoming query source addresses. We use network tunnel (TUN) to redirect all the DNS queries and responses to proxies. Those proxies further manipulate packets addresses to successfully deliver the packets and to let the meta-DNS-server find the correct answers (Figure 2). We explain details of our solutions in the following.

To redirect recursive server's queries to meta-DNS-server we must change the destination or source addresses of those DNS packets.

Before any address manipulation, we first need to capture *all* the queries and responses, because any leaked packets is non-routable and dropped, leading to the failure of trace replay. We create two TUN interfaces to get all required packets at the recursive and meta-DNS-server respectively (Figure 2). We use *port based routing* that all queries (packets with destination port 53) at the recursive, and responses (packets with source port 53) at the meta-DNS-server are routed to TUN interfaces. We manage this routing by using `iptable`: first mark the desired packets using `mangle` table, and then redirect all the marked packets to TUN interfaces. We choose TUN interface because it let us observe all raw IP packets to manipulate IP addresses.

We build two proxies (*recursive proxy* and *authoritative proxy*) to manipulate packet addresses at the recursive server and meta-DNS-server respectively (Figure 2). The common task of the proxies is to make sure captured packets can be routed to the server at the other end smoothly for correct trace replay. Specifically, recursive proxy captures recursive server's queries and authoritative proxy captures meta-DNS-server's responses. Then, both of the proxies rewrite the destination address with the IP address of the server at the other end.

To make the meta-DNS-server determine the correct answer and let the recursive server accept the reply, the proxies replace the source address with the original destination address in the packets. We will explain the functionality of using original destination address below. After recalculating the checksum, the proxies send the modified packets directly to the meta-DNS server and the recursive server respectively.

This process with proxy rewriting allows the meta-DNS server to determine to which zone each query is addressed. To address the zone selection, the meta-DNS server hosts multiple zones using software-based split-horizon DNS [1], where a server provides different answers based on query source and destination addresses. When a recursive server resolves a incoming query iteratively with cold cache, the destination addresses (target authoritative server address) of the iterative queries is the only identifier for different zones, because the query content is always the *same* and not distinguishable by itself. However, matching queries by destination addresses at the meta-DNS-server requires the server listens on different network interfaces for each zone separately, which brings deployment complexity, such as creating many (virtual) network interfaces and a giant routing table in testbed. This complexity conflicts our goal of scalability and deployability to support many different zones.

With split horizon, the meta-DNS server listens on one address and uses the source IP address to determine for which level of the hierarchy the query is destined. Since recursive proxy already replaces the query source address with the original query destination address (OQDA), the current query source address becomes the zone identifier now. To correctly discriminate queries for different zones, we take the public IP addresses of zone's nameservers as the matching criteria (query source addresses). In this way, the meta-DNS-server sees a query coming from OQDA instead of the recursive server's address (Figure 2). The meta-DNS server then determines the correct zone file from the this source address, and issues a correct reply where the destination address is OQDA. As discussed above, the authoritative proxy captures this reply, and puts the destination address in source address. As a result, the recursive server observes a normal reply from OQDA and can match this reply to the original query, without knowing any address manipulation in the background. Our method works with authoritative server implementation that supports split-horizon DNS, such as `BIND` with its `view` and `match-clients` clauses in configuration.

2.5 Mutate Trace For Various Experiments

Another benefit of our system is that we support arbitrary trace manipulation to study different questions from one trace.

There are two challenges to change the traces. First, binary network trace is complicated to edit directly because changes are not space-equivalent. We need a user-friendly method to manipulate queries. Second, the delay caused by manipulation and processing traces, may also bring problems for accurate query replay.

Plain text for easy manipulation: To easily manipulate input queries, we convert network traces to human-readable plain text. We develop a DNS parser to easily extract relevant data from network trace, and output a column-based plain text file where each line contains necessary information of a DNS message. In this stage, users can edit DNS messages as desired.

Binary for fast processing: Since plain text as input delays building DNS messages, we convert the resulting text file to a customized binary stream of internal messages to serve as input for trace replay (Figure 3) for fast processing. To distinguish different messages in the input stream, we pre-pend the length of each message at the beginning of each binary message.

To save unnecessary input delay in query replay, we pre-process the input and separate the input processing from the query replay system. Optionally, the input engine of our system can also read network trace and formatted text file directly, and convert to internal binary messages on the fly.

2.6 Distribute Queries For Accurate Replay

With server setup and input trace, the next step for a successful DNS trace replay is to emulate DNS queries with *correct timing* from different sources and connections.

Fast query replay and diverse sources: There are several resource limit in a single host: CPU, memory and the number of ports. The query rate generated at a single host is limited because of CPU constraints. The ability to maintain concurrent connections in a single host is limited by memory and the number of ports (typical 65 k).

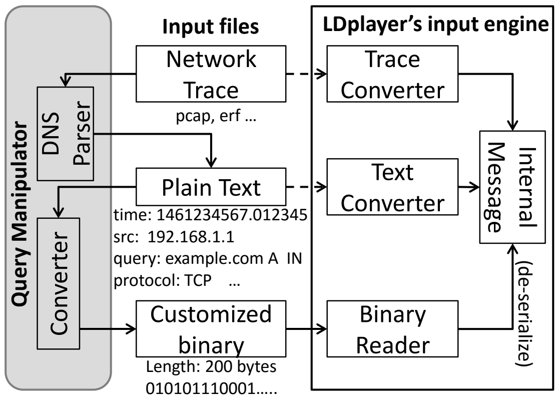


Figure 3: Trace manipulator converts network trace to plain text for easy editing, and further converts to customized binary stream as input. LDplayer accepts three types of input: network trace, formatted plain text and customized binary files.

To support fast query rates from many sources, our approach is to distribute query stream to many different hosts to allow many senders to provide a large aggregate query rate In particular, we coordinate queries from many hosts with a central *Controller* managing a team of *Distributors* which further controls several *Queriers*. The end Queriers directly interact with DNS servers via UDP or TCP. For reliable communication, we decide to choose TCP for message exchange among distributors.

The primary purpose of multiple levels is to connect enough end Queriers when there is a limit on the number of distribution connections in each Distributor. Without limit, theoretically one-level distribution (Controller distributes to Queriers directly) can bring 4 billion connections in total, with maximum 65 k Querier hosts connected at any time.

If the input trace is extremely fast, the CPU of Controller may become bottleneck because it limits the speed of input processing. To solve this problem, we can split input stream to feed multiple controllers.

Correct timing for replayed queries: The ultimate goal of query replay system is to replay DNS queries with correct timing and reproduce the traffic pattern.

Due to distributing queries among different hosts, it is challenging to synchronize time and ensure the correct timing and ordering of individual queries.

To replay queries at accurate time, LDplayer keeps tracking trace time and real time, and schedules *timer events* to send queries. When getting the first input query message, controller broadcasts a special *time synchronization* message to *all* the queriers to indicate the start time of the trace. Upon receiving the time synchronization message, a querier obtains the query time in the trace ($\bar{t}_1$) and the current real time (t_1).

After time synchronization message, controller starts to distribute input query stream based on the strategy of distribution by sources discussed above. On receiving a query message (q_i), a querier extracts current absolute query time in trace ($\bar{t}_i$) and computes the relative trace time ($\Delta\bar{t}_i$), as $\Delta\bar{t}_i = \bar{t}_i - \bar{t}_1$.

The relative trace time is the ideal delay that should be injected for trace replay assuming no input delay. Similarly, the querier also gets current absolute real time (t_i) and the relative real time (Δt_i) as $\Delta t_i = t_i - t_1$. The relative real time represents the accumulated program runtime delay, such as input processing and communication delay, that has already been generated.

To replay the query (q_i) at correct time, LDplayer removes the added latency and schedules a timer event at ΔT_i in the future, where $\Delta T_i = \Delta\bar{t}_i - \Delta t_i$.

By tracking timing and continuously adjusting, LDplayer provides good absolute and relative timing (as shown in §4). If the trace is extremely fast and the input processing falls behind ($\Delta T_i \leq 0$), LDplayer sends the query immediately without setting up a timer event.

Some experiments, such as load testing, prefer *large* query streams, as *fast* as possible, instead of tracking original timing time. As an option, LDplayer can disable time tracking and replay as fast as possible.

Emulating queries from the same source: Some traces or experiments require reproduction of inter-query dependencies. Two examples are UDP queries where the second query can be sent only after the first is answered, or when studying TCP queries where connections are reused. In general, we assume all queries from the same source IP address are dependent and queries from different sources are independent.

We use different *network sockets* to emulate query sources. To emulate queries from the same sources, we must first deliver all the queries from the same sources (IP addresses) in the original trace to the same end querier for replay.

To accomplish this, each distributor tracks the original query source address and the lower level component in the message distribution flow. When queries are distributed, each distributor either picks the next entity based on a recent query source address in record, or selects randomly otherwise (during startup). Each entity keeps the record during the experiments.

Similarly, queriers map the query sources and the underlying *network socket*, insuring that same-source queries use the same socket if it is still open. New sources start new sockets.

When emulating TCP connection reuse, queriers also track of open TCP connections. They may close them after a pre-set timeout.

As a result, during query replay, a DNS server ob-

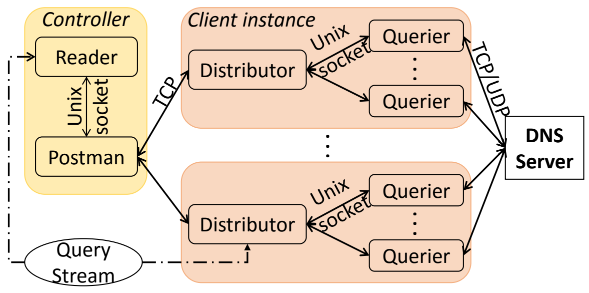


Figure 4: A prototype of distributed query system with two-level query distribution. Distributors and queriers are implemented as processes and running on the same host (*client instance*). Optionally, a single distributor can read input query stream directly.

serves queries from the same set of host addresses but with a range of different port numbers, which emulates different queries from the same sources.

An alternative is to setup virtual interfaces with different IP addresses at queriers, and use those interfaces for each query sources address in query replay. However, the method does not scale to a large number of addresses.

3 Implementation

We implement our prototype replay system and proxies in C++, to provide efficient runtime and controlled memory usage.

Query System: In two-level query distribution system (Figure 4), with a controller and multiple clients. The controller runs two processes, the *Reader*, for trace input, and another, the *Postman* to distribute queries. One or more machines are clients, each with distributor and multiple querier processes. Processes use event-driven programming to minimize state and scale to a large number of concurrent TCP connections. The reader pre-loads a window of queries to avoid falling behind real time.

Server Proxy: The proxies around the server run as either recursive proxy or authoritative proxy (§2.4). A single *reader* thread reads from a tunnel network interface, while multiple worker threads read from a thread-safe queue that rewrites queries (§2.4).

4 Evaluation

We validate the correctness of our system by replaying different DNS traces in controlled testbed environment (§4.1). Our experiments show that the distributed client system replays DNS queries with correct timing, reproducing the DNS traffic pattern (§4.2).

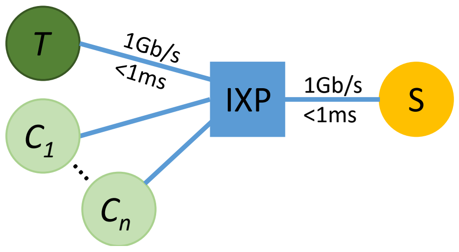


Figure 5: Network topology for experiment: controller (T), server (S), and client instances (C) running with distributor and querier processes.

4.1 Experiment Setup and Traces

To evaluate our system, we deploy the network shown in Figure 5 in the DETER testbed [4]. We use a controller (T) to distribute query stream to client instances (C_1 to C_n). Each client instance runs several distributor and querier processes to replay input queries. The query traffic merges at a LAN representing an Internet Exchange Point, and is then sent to the server (S). Each hosts is a 4-core (8-thread) 2.4 GHz Intel Xeon running Linux Ubuntu-14.04 (64-bit). We use several traces, listed in Table 1 and described below, to evaluate the correctness of our system under different conditions.

B-Root: This trace represents all traffic at B-Root DNS server over one hour during the 2016 and 2017 DITL collections [8]. It is available from the authors and DNS-OARC. We use B-Root-16 trace (Table 1) in this section to validate our system can accurately replay high-volume queries against an authoritative server. We use other groups of B-Root-17 traces in later sections (§5).

Synthetic: To validate the capability to replay query traces with various query rates, we create five synthetic traces (syn-0 to syn-4 in Table 1), each with different, *fixed* inter-arrival times for queries, varying from 0.1 ms to 1 s. Each query uses a unique name to allow us to associate queries with responses after-the-fact.

4.2 Accuracy of Replay Timing and Rate

We first explore the accuracy of the timing and rate of query replay.

Methodology: We replay B-Root and synthetic traces over *UDP* in real time and capture the replayed traffic at server. We match query with reply by prepending a unique string to every query names in each trace. We then report the *query timing*, *inter-arrival time* and *rate*, comparing the original trace with the replay. We use a real DNS root zone file in server for B-Root trace replay to provide responses. For synthetic trace replay, we setup the server to host names in `example.com` with wildcards, so that it can respond all the queries within

traces	start	(min)	inter-arrival (seconds)	client IPs	records
B-Root-16	2016-04-06 15:00 UTC	+60	.000027 ±.000619	1.07 M	137 M
B-Root-17a	2017-04-11 15:00 UTC	+60	.000023 ±.001647	1.17 M	141 M
B-Root-17b		+20	.000025 ±.001536	725 k	53 M
Rec-17	2017-09-01 17:22 UTC	+60	.180799 ±.355360	91	20 k
Synthetic					
syn-0	-	60	1	3 k	3.6 k
syn-1	-	60	.1	9.7 k	36 k
syn-2	-	60	.01	10 k	360 k
syn-3	-	60	.001	10 k	3.6 M
syn-4	-	60	.0001	10 k	36 M

Table 1: DNS traces used in experiments and evaluation. Mean and standard deviation of inter-arrival time for B-Root and Rec traces.

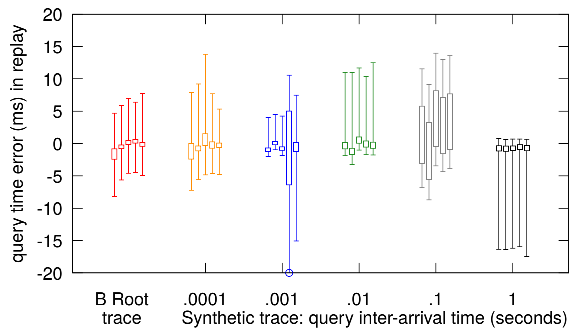


Figure 6: Query timing difference between replayed and original traces. Figure shows quartiles, minimum and maximum. The empty circles on x-axis exceed ± 20 ms (outliers).

that domain. We repeat each type of trace replay for 5 times to avoid outliers.

Query time: We use unique query names to identify the same queries in original and replayed traces, and study the timing of each query: the absolute time difference compared to the first query. We ignore the first 20-seconds of the replay to avoid startup transients.

Figure 6 shows that timing differences in replay are tiny, usually quartiles are within $\pm 2.5 ms$. We observe small, but noticeably larger differences when the query interarrival is fixed at 0.1 s: $\pm 8 ms$ quartiles. We are examining this case, but suggest it is an interaction between application and kernel-level timers at this specific timescale. Even when we look at minimum and maximum errors, timing differences are small, within $\pm 17 ms$.

Query Inter-arrival Time: We next shift from absolute to relative timing with inter-arrival times.

Figure 7 shows the CDF of experimental interarrival

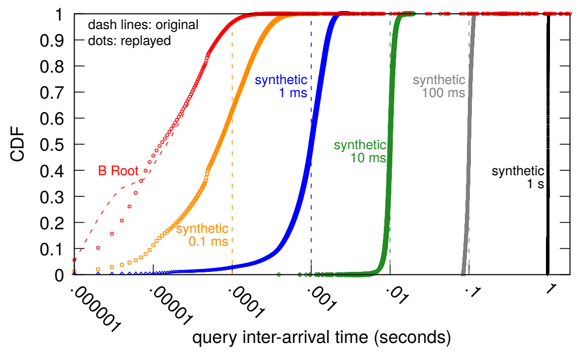


Figure 7: Cumulative distribution of the inter-arrival time of original and replayed traces.

times for real (B-Root-16) and synthetic traces of different interarrival rates. (Note that timescale is shown on a logarithmic scale.) Interarrival is quite close for traces with input inter-arrivals of 10 ms or more, and for real-world traffic with varying interarrivals. We see larger variation for very small, fixed interarrivals (less than 1 ms), although the median is on target, there is some variation. This variation occurs because it is hard to synchronize precisely at these fine timescales, since the overhead from system calls to coordinate take nearly as much time as the desired delay, adding a lot of jitter. We see divergence for the smallest interarrivals for the real-world B-Root trace, but little divergence for the 50% longest B-Root interarrivals. Uneven spacing in real traces gives us fee time to synchronize. We repeat this experiment for 5 times; all show similar results to the one shown here.

Query Rate: We finally evaluate query rates. To do so, we replay the B-Root-16 trace and compute the query rate in each second of trace replay against the corresponding rate of that second in the original trace. We repeat this test five times.

Figure 8 shows the CDF of the difference in these per-second rates for all 3,600 seconds of each of the five replays. We observe that almost all (4 trials with 98%-99% and 1 trial with 95%) of 3.6 k data points (1-hour period) have tiny ($\pm 0.1\%$) difference in average query rate per second. This experiment uses the B-Root because it has large query rate (median 38 k queries/s) and the rate varies over time. We use a 1-second window to study overall replay rate; finer (smaller) windows may show greater variation as OS-scheduling variation becomes significant.

4.3 Single-Host Throughput

Having shown our query system is accurate to replay different traces, we next evaluate the maximum throughput:

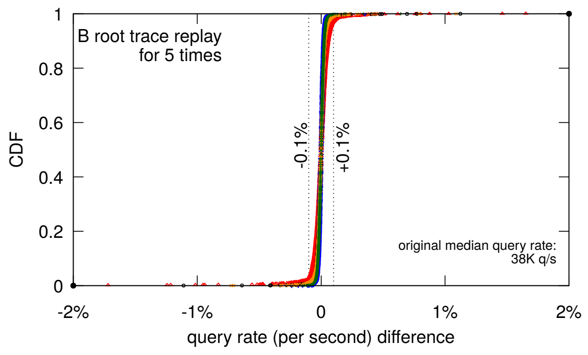


Figure 8: Query rate differences between replayed and original B-Root trace (5 trials). Black solid circles on the edge are a few cases out of ±2%.

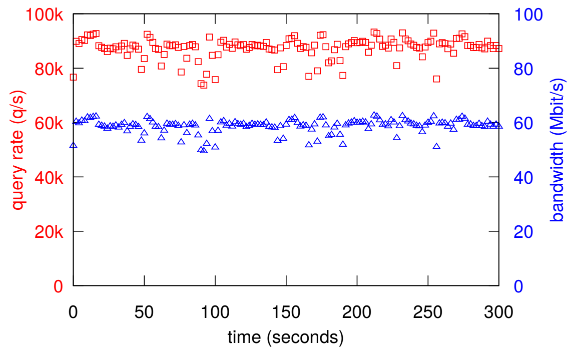


Figure 9: The throughput of *fast* replay a continuous input query stream over UDP directly: queries are sent immediately without timer events. Data point is sampled every two seconds over total 5 minutes.

how fast can our system replay using a *single* host?

Methodology: We use an artificial query generator for controlled, high-speed replay. We send a continuous stream of identical queries (`www.example.com`) to the target, sending them with UDP, without timeouts, to an authoritative server hosting `example.com` zone with wildcards. We run our query replay system with one distributor and six querier processes, along with the query generator (total 8 processes), on a single 4-core (8-hyperthread) host. We monitor the packet rate and bandwidth after the query system is in steady state.

Results: With this setup we replay 87 k queries/s (60 Mb/s), as shown in Figure 9). This rate is more than twice of normal DNS B-Root traffic rate (as of mid-2017). In this experiment the query generator is the bottleneck (it completely saturates one CPU core), while other processes (distributor and queriers) each consumes about 50% of single CPU core. Higher rates would be possible with faster query generation.

5 Applications

With controlled, configurable and accurate trace replay, our system provides a basis for large-scale DNS experimentation which further enables real world applications to answer open research questions. We next show example applications of LDplayer, including studying the impact of increased DNSSEC queries and exploring the performance of DNS over TCP at a DNS Root server.

5.1 Impact of Increased DNSSEC Queries

How does the root traffic change when more and more applications start to use DNSSEC? We start to answer this question and predict future DNS root traffic by replaying traffic with a mix of different key sizes and different portions of queries with DNSSEC. With the same experiment setup (§4.1), we use LDplayer to change DNSSEC OK (DO) bit in queries and replay B-Root query traffic (Table 1).

We observe that going from 72% DO (today) to 100%, root response traffic becomes 296 Mb/s (median) with 2048-bit ZSK in steady state (Figure 10). Compared to 225 Mb/s with current 72% DO and 2048-bit ZSK, root response traffic could increase by 31% in the future when all queries require DNSSEC. Our experiments also demonstrate 32% traffic increase when DNS root ZSK was upgraded to 2048-bit from 1024-bit keys, replicating experiments previously done in a custom testbed [21]. As a future work, we could use LDplayer to study the traffic under 4096-bit ZSK.

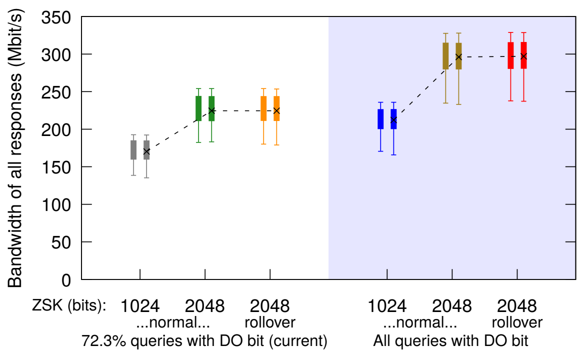


Figure 10: Bandwidth of responses under different DNSSEC ZSK sizes. Trace: B-Root-16. Figures show medians, quartiles, 5th and 95th percentiles.

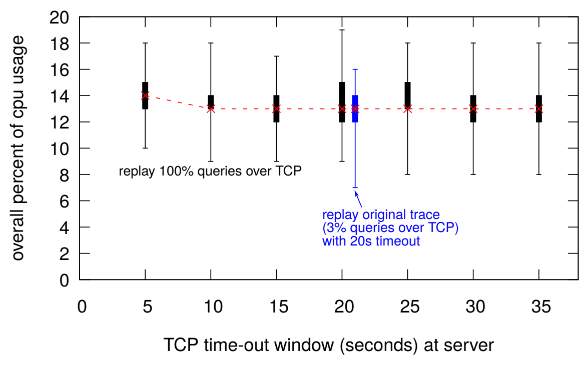


Figure 11: CPU usage with different TCP timeouts under minimal RTT (<1 ms). Trace: B-Root-17a. Figures show medians, quartiles, 5th and 95th percentiles.

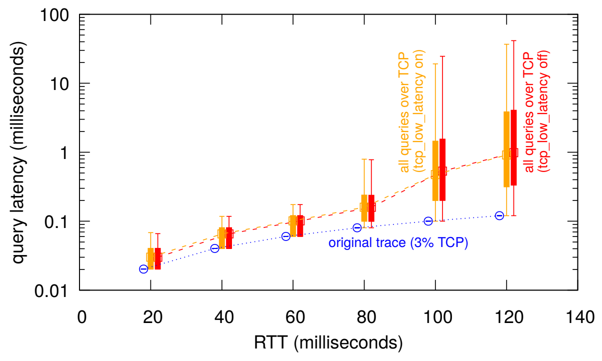


Figure 12: Query latency with *20-second* TCP timeout and different RTTs. Trace: B-Root-17b. Figures show medians, quartiles, 5th and 95th percentiles.

5.2 Performance of DNS over TCP at a Root Server

We next use experiments to study DNS over TCP, and determine the effects of such a change of resource usage (memory and CPU) and latency. These topics have previously been studied with micro-benchmarks and modeling [24], but our experiments here are the first to study them at scale with a full server implementation.

5.2.1 Experiment Setup and Methodology

To evaluate server resource requirement and query latency, we deploy the network topology (Figure 5 and §4.1). We vary the client-to-server RTT for different experiments. All client hosts use 16 GB ram and 4-core (8-thread) 2.4 GHz Intel Xeon. To support the all-TCP workload, we configure the authoritative server 24 GB RAM on a 12-core (24-thread) 2.2 GHz Intel Xeon, using `nsd-4.1.17` with 8 processes. All hosts run Ubuntu-16.04 (64-bit).

We conduct two types of query replay using B-Root-17 traces (Table 1). First, we replay the queries using the protocols in the original trace (3% TCP queries) to establish a baseline for comparison. We then mutate the queries so *all* employ TCP. We vary either the client-server RTTs (0 ms to 120 ms) or TCP timeouts (5 ms to 35 ms) at the server.

We use two B-Root traces in the experiments in this section. We first use use 1-hour B-Root-17a trace to study server state with controlled minimal RTT (<1 ms), verifying the experiment reaches steady state in about 5 minutes. For later experiments we use B-Root-17b, a 20-minute subset of the B-Root-17a trace.

We log server memory with `top` and `ps`, CPU with `dstat`, and active TCP connections with `netstat`.

5.2.2 Memory and Connection Footprint

For DNS over TCP, a server should keep idle connections open for some amount of time, to amortize TCP connection setup costs [12, 24]. However, a server cannot keep the connection open forever, since maintaining concurrent connections costs server memory.

We first evaluate server memory and connection requirement. Figure 13 shows the memory and connection footprint in experiments of replaying original traces and all queries over TCP with different connection timeouts. We demonstrate that both the number of active TCP connections and server memory consumption rise as the TCP timeout increases. We show that with 20 s TCP timeout suggested in prior work [24], our experimental server requires about 13 GB memory (Figure 13a) and 180 k connections, one-third are active (Figure 13c) and the rest in TIME_WAIT state (Figure 13b). These values are well within current commodity server hardware. Resource usage reaches steady state in about 5 minutes and is thereafter stable (approximately flat lines in Figure 13).

By contrast, we observe that the server only needs about 2 GB memory (blue bottom line in Figure 13a), when replaying the original trace (3% TCP queries) at 20 s timeout. DNS operators with old hardware will need to upgrade server when preparing for DNS over TCP.

We expected memory to vary depending on querier RTT, but the memory and CPU usage do not change regardless of the distance from client to server (figure provided in Appendix A). This resource stability is because memory and CPU are dominated by connection timeout duration, which at 20 s is 200× longer than RTTs.

Our experimental results confirm prior models in a real-world implementation, showing that even if all DNS were shifted to TCP, memory and CPU requirements are manageable (13 GB with 20 s connection timeout), although much larger than today's UDP-dominated DNS.

5.2.3 CPU Usage

We next evaluate server CPU usage for DNS over TCP.

Figure 11 shows the statistics of server CPU usage. We observe that overall the CPU usage is about 10% to 20% over 24 cores for all queries over TCP, again manageable on current commodity server hardware. Results are stable regardless of the connection timeout window (the flat red line). We observe a slightly higher (1% more at median) CPU usage at 5 ms timeout, likely due to more frequent connection timeout and setup.

In contrast, replaying original trace (3% TCP queries) with 20 ms TCP timeouts has slightly (2%) lower at maximum and minimum CPU usage, although the median CPU usage is the same as replaying all queries over TCP (blue bar in Figure 11).

Our experiments confirm that connection tracking in TCP does not increase CPU usage noticeably over UDP. This result was only possible in experiment, since there are no good models of CPU consumption for DNS.

5.2.4 Query Latency

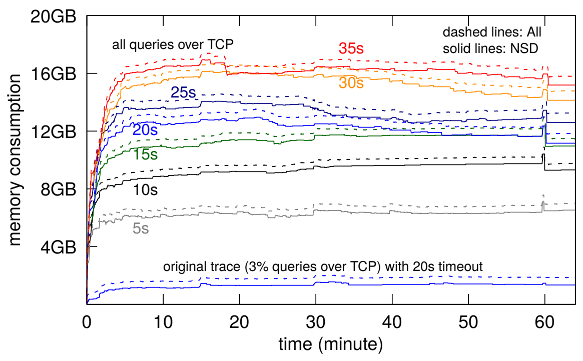
We finally evaluate query latency for DNS over TCP. Figure 12 shows the statistics of query latency with different RTTs. TCP connection reuse helps to reduce query latency: median query latency in TCP is only about 50% to 60% slower than UDP, while if all connections were fresh, models predict 100% overhead due to the extra RTT in connection setup.

Experimentation also helps reveal differences due to RTT—as the RTT increases, the query latency of TCP become much larger than UDP latency. For example, latency of all query over TCP is 7-time more (by median) than original trace (3% TCP) replay at large 120 ms RTT. We observe that TCP query latency reduces (70 ms less by median at 100 ms vs 120 ms RTT), by enabling `net.ipv4.tcp_low_latency` in Linux to avoid adding latency. However, this optimization does not change query latency under smaller RTTs (<80 ms).

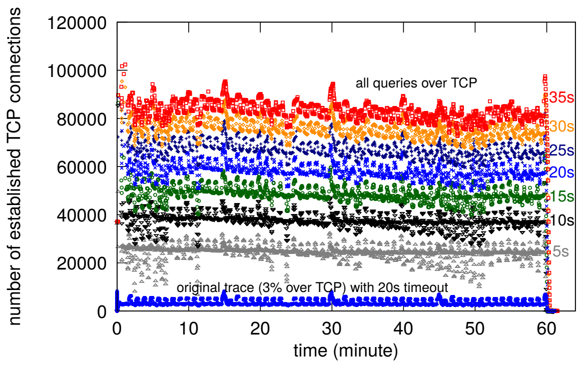
The multi-time RTT latency of TCP queries is unexpected since a single TCP query would only require 2 RTTs. By examining packet traces, we see many server reply TCP segments (possibly DNS messages) reassembled into a large TCP message, even with `net.ipv4.tcp_low_latency` enabled. Resembling may cause the large delay in DNS over TCP, because waiting for all the packets. Another optimization is to disable the Nagle algorithm on the server.

By contrast, latency with UDP is consistent regardless of RTT, because UDP has no algorithms like Nagle trying to reduce packet counts.

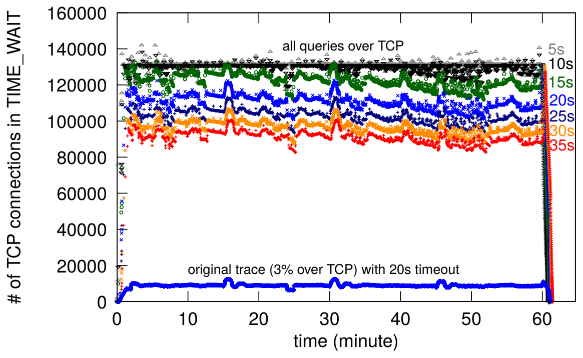
Evaluating these real-world performance interactions between the DNS client and server was only possible in full trace-driven experiments, since there no generic model for TCP query processing in DNS servers. Our



(a) Memory consumption.



(b) Established TCP connections.



(c) TCP connections in TIME_WAIT state.

Figure 13: Evaluation of server memory and connections requirement with different TCP timeouts and minimal RTT (<1 ms). Trace: B-Root-17a

experiments shows the effect of TCP connection reuse although the TCP query latency is still noticeably larger than UDP, providing much greater confidence to testbed experiments with synthetic traffic and modeling [24]. Our use of real traces and server software also showed an unexpected increase in TCP query latency for large client RTTs.

6 Related Work

DNS Replay Systems: Several other systems that replay DNS traffic and simulate parts of DNS hierarchy. Wessels et al. simulate the root, TLD and SLD servers with three computers to study the caching effects of different resolvers on the query load of upper hierarchy [22]. Yu et al. build a similar system with multiple TLD servers hosting one TLD (`.com`), to understand authority servers selections of different resolvers [23] . Ager et al. set up a testbed simulating DNS hierarchy to study DNSSEC overhead [2]. DNS-OARC develops a DNS traffic replay tool [7] to test server load.

Our system differs from these in scale, speed, and flexibility. Each of these systems host each zone on a different name server, so they cannot scale to thousands of zones. They also often make modifications to the zones (dropping and modifying `NS` records), to make the routing work and obtain the correct answers from servers. We instead use proxies to allow all zones to be provided from one name server, and to provide a query sequence that matches real DNS. In addition, these systems do not carefully track timing. (For example, the Ager et al. system uses batch-mode `dig` and so can handle only light loads.) Our client system replays DNS queries with correct timing, reproducing the traffic pattern accurately. Finally, prior systems are designed to recreate today's protocol; we instead include the ability to project a current trace through future protocol options, such as replaying UDP queries as TCP with preset connection timeout.

Traffic Generators: Several traffic generators can create DNS [9, 16]. Like these tools, our query replay system can also generate a stream of DNS packets with specified parameters. However, these tools are not specific for DNS; they provide only simple replay or generation. Our system focuses on DNS protocol and provides a generic DNS experimentation platform. Our system can replay queries with accurate timing, and mutate queries to test what-if scenarios.

General Network Replay: Several tools replay general network traces [15, 20, 10]. While these tools can replay DNS trace with timing given in the trace, our replay-client system simulates the DNS query semantics, allowing us to replay real-world queries with different variations (such as if all used TCP). Rather than just replaying each packet in the trace mechanically, our system allows exploration of future DNS design options.

DNS Studies: There are studies that replay DNS queries to evaluate the performance of DNS applications [18, 13, 6]. Our replay-client system supports analysis like these studies, but it provides a more flexible platform that also enables *new* studies at high query rates with protocol variants.

To the best of our knowledge, ours is the only experimental DNS system that can replay DNS trace with original zone files, uses distributed clients to handle large query rate and simulate different query sources, and lets us vary protocols.

7 Conclusion

This paper has described LDplayer, a system that supports trace-driven DNS experiments. This replay system is efficient (87k queries/s per core) and able to reproduce precise query timing, interarrivals, and rates (§4). We have used it to replay full B-Root traces, and are currently evaluating replays of recursive DNS traces with multiple levels of the DNS hierarchy.

We have used our system to evaluate alternative DNS scenarios, such as where all queries use DNSSEC, or all queries use TCP. Our system is the first to make at-scale experiments of these types possible, and experiments with TCP confirm that memory and latency is good (as predicted by modeling), but highlight performance variation in latency due to implementation details not captured in models. In addition, experimental confirmation of complex systems factors such as memory usage are critical to gain confidence that an all-TCP DNS is feasible on current server-class hardware.

References

[1] Split-horizon dns. `https://en.wikipedia.org/wiki/Split-horizon_DNS`.

[2] Ager, B., Dreger, H., and Feldmann, A. Predicting the DNSSEC overhead using DNS traces. In *Annual Conference on Information Sciences and Systems* (March 2006), pp. 1484–1489.

[3] Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFCs 6014, 6840.

[4] Benzel, T. The science of cyber security experimentation: The deter project. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 137–148.

[5] Bortzmeyer, S. DNS privacy considerations. RFC 7626, Aug. 2015.

[6] Brustoloni, J., Farnan, N., Villamarín-Salomón, R., and Kyle, D. Efficient detection of bots in subscribers' computers. In *Communications, 2009. ICC'09. IEEE International Conference on* (2009), IEEE, pp. 1–6.

[7] DNS-OARC. drool. `https://github.com/DNS-OARC/drool`.

[8] DNS-OARC. Day In The Life of the internet (DITL) 2017. `https://www.dns-oarc.net/oarc/data/ditl/2017`, Apr. 2017.

[9] Haas, H. Mausezahn. http://netsniff-ng.org/.

[10] Heng, A. Y. C. Bit-twist. http://bittwist.sourceforge.net/.

[11] Hoffman, P., and Schlyter, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012. Updated by RFCs 7218, 7671.

[12] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and Hoffman, P. Specification for DNS over Transport Layer Security (TLS). RFC 7858 (Proposed Standard), May 2016.

[13] Khurshid, A., Kiyak, F., and Caesar, M. Improving robustness of dns to software vulnerabilities. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 177–186.

[14] Lewis, C., and Sergeant, M. Overview of Best Email DNS-Based List (DNSBL) Operational Practices. RFC 6471 (Informational), Jan. 2012.

[15] Loef, A., and Wang, Y. libtrace tool: tracereplay. http://www.wand.net.nz/trac/libtrace/wiki/TraceReplay.

[16] Nathan, J. nemesis. http://nemesis.sourceforge.net/.

[17] Osterweil, E., Ryan, M., Massey, D., and Zhang, L. Quantifying the operational status of the dnssec deployment. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2008), IMC '08, ACM, pp. 231–242.

[18] Park, K., Pai, V. S., Peterson, L. L., and Wang, Z. Codns: Improving dns performance and reliability via cooperative lookups. In *OSDI* (2004), vol. 4, pp. 14–14.

[19] Su, A.-J., Choffnes, D. R., Kuzmanovic, A., and Bustamante, F. E. Drafting behind akamai (travelocity-based detouring). In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2006), SIGCOMM '06, ACM, pp. 435–446.

[20] Turner, A., and Klassen, F. Tcpreplay. http://tcpreplay.appneta.com/.

[21] Wessels, D. Increasing the Zone Signing Key Size for the Root Zone. In *RIPE 72* (May 2016).

[22] Wessels, D., Fomenkov, M., Brownlee, N., and claffy, k. Measurements and Laboratory Simulations of the Upper DNS Hierarchy. In *Passive and Active Network Measurement Workshop (PAM)* (Antibes Juan-les-Pins, France, Apr 2004), PAM 2004, pp. 147–157.

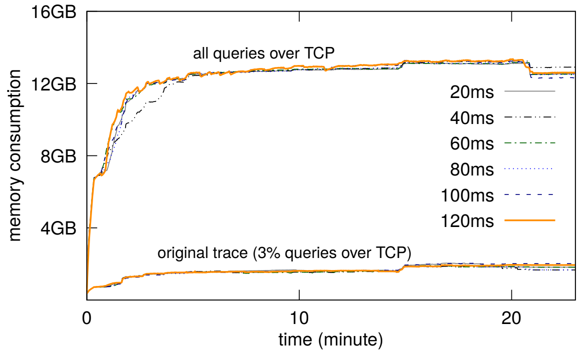
[23] Yu, Y., Wessels, D., Larson, M., and Zhang, L. Authority server selection in dns caching resolvers. *SIGCOMM Comput. Commun. Rev. 42*, 2 (Mar. 2012), 80–86.

[24] Zhu, L., Hu, Z., Heidemann, J., Wessels, D., Mankin, A., and Somaiya, N. Connection-oriented dns to improve privacy and security. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 171–186.

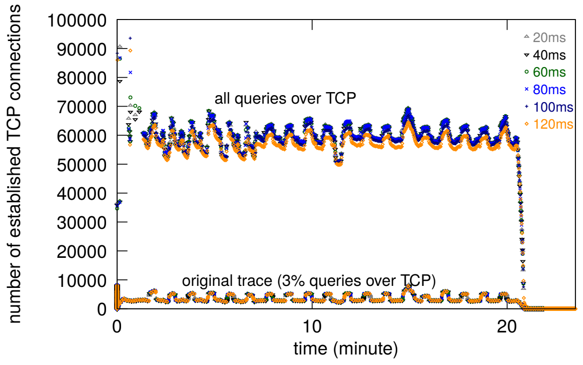
[25] Zhu, L., Wessels, D., Mankin, A., and Heidemann, J. Measuring DANE TLSA deployment. In *Proceedings of the 7th IEEE International Workshop on Traffic Monitoring and Analysis* (Barcelona, Spain, Apr. 2015), Springer, pp. 219–232.

A TCP Performance As A Function of RTT

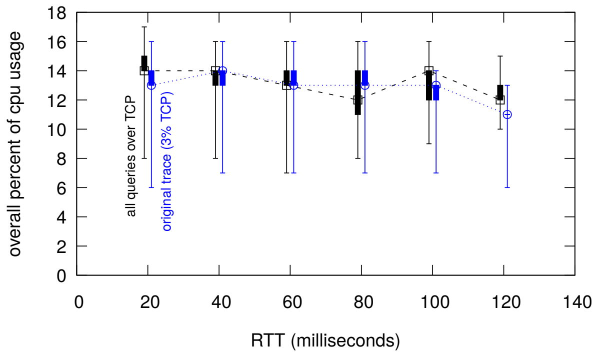
We expected memory to vary depending on querier RTT, but the memory and CPU usage do not change regardless of the distance from client to server, as shown in Figure 14. This resource stability is because memory and CPU are dominated by connection timeout duration, which at 20 s is 200× longer than RTTs.



(a) Memory consumption.



(b) Established TCP connections.



(c) CPU usage

Figure 14: Evaluation of server resource by replaying traces with *20-second TCP timeout* and different RTTs. Trace: B-Root-17b