Catamaran: A Raft-Based Fault-Tolerant and Distributed DNS Nameserver

Kamran Ahmed
Stanford University

Jeremy Kim
Stanford University

Hari Vallabhaneni
Stanford University

Ruiqi Wang
Stanford University

ABSTRACT

We present a fault-tolerant and scalable distributed DNS nameserver that replicates DNS resource records using our own custom (simplified) implementation of Raft. Our nameserver supports Dynamic DNS (DDNS) updates where services can update resource records without the need to manually edit zone configuration files. We present a motivating example where services can continuously monitor their public IPv4 address and push new resource records to our name server with immediate effect. This grants clients uninterrupted access to services using their static domain names even in dynamic environments where host IP addresses may change often. We measure query and update latency, our system's ability to tolerate node failures, and the overall cost of resource record replication compared to BIND 9, a popular DNS nameserver software.

1 INTRODUCTION

The Domain Name System (DNS) service forms an important and highly utilized backbone of Internet services, mapping their human-readable domain names, which tend to stay the same, to their Internet Protocol (IP) addresses, which may change over time. These domain names are used to access important resources, such as websites and mail servers. Without DNS, the Internet would become significantly less usable for end-users.

DNS nameservers, which are responsible for responding to queries regarding a particular domain name's resources, must endure failures, provide low latency for IP address resolution, and handle many concurrent queries. A nameserver's failure would prevent access to the services associated with its domain, such as a website hosted at that domain [15]. Furthermore, since DNS querying is often the first step to connecting to an Internet service, poor performance at the DNS nameserver, either due to high latency or low throughput of DNS queries, would result in a bottleneck for starting the connection. Due to the massive scale of Internet traffic, the suboptimal performance of important nameservers could potentially make popular services unusable for at least thousands of users.

This work specifically investigates nameserver performance with respect to Dynamic DNS (DDNS). DDNS allows Internet services to remain accessible even when their IP addresses change continuously by facilitating frequent writes to a domain's nameserver, such as updating the IP address of a subdomain [2]. Although large services often operate from a static set of IP addresses, there are multiple situations where a service's IP address would change frequently, thus necessitating studies on DDNS. First, smaller services may rely on ISP or cloud-issued ephemeral IP addresses. Second, a mobile service's IP address changes as it physically moves between subnets. Major providers, such as Amazon [10] and Cloudflare [2], support DDNS, further emphasizing its relevance.

DDNS imposes additional requirements on a performant DNS nameserver. First, low latency and high throughput for reads become paramount. Resource records, which store information regarding a domain's IP addresses, cannot be cached by an end-user for long periods because they may become stale. Thus, the DDNS service must support frequent and fast reads to be performant. Second, these services must support an acceptable write throughput and latency. While most DNS records are rarely updated because the domain uses static IPs, DDNS records may be updated at a non-negligible rate—for instance, an ISP may decide to re-issue IP addresses at a certain time to all of its customers, or several mobile or IoT devices hosting services may obtain new DHCP leases or cross networks at the same time.

How can we develop a DDNS service that is fault-tolerant, fast, and capable of handling many concurrent requests? What are the performance tradeoffs incurred through replication? In this work, we develop a DDNS service that is replicated across 5 servers with a custom implementation of the Raft consensus algorithm [11]. We evaluate its throughput and latency for reads and writes under normal and faulty conditions and compare our service's performance to a load-balanced BIND 9 DNS nameserver [1], an established software for operating nameservers.

2 BACKGROUND AND PREVIOUS WORK

Existing implementations of replicated or redundant DNS nameservers have been widely used for several years already, due to the reasons discussed above. One common approach is anycast-based [9] distribution, where "one-to-many" client connections are multiplexed among multiple equally privileged nameservers that share a single logical external IP address. While this approach provides good load balancing and enables users to continue sending queries in the event that one or a few servers go down, there is no built-in mechanism to maintain consistent data among nameservers, and supporting anycast routing complicates deployment.

Another approach separates responsibilities between primary and secondary nameservers [13, 14], where the primary server is the central source of zone files, and has the ability to update resource records; it propagates updates to secondary servers. Secondary nameservers are able to resolve client queries, but they cannot update records. Still, this scheme provides high read availability in that services are still accessible if the primary server goes down. We implement this approach with BIND.

Consensus-based replication is a natural next step for achieving read and write fault tolerance; when the leader fails, a follower can immediately step up and maintain write availability for clients. Load-balancing functionality can easily be added as an extra layer that sits between the client and the replicated cluster. Although the remaining properties of high throughput and low latency are not immediate, consensus algorithms remain an interesting direction for improving nameserver performance. This work investigates the Raft consensus algorithm's applicability to DDNS.

The Raft consensus algorithm is a leader-based consensus algorithm. It achieves consistent state among a cluster of peer servers by electing a server to act as a leader. All writes to the shared state go to the leader, and the leader periodically disseminates these updates to the rest of the cluster, its followers, by sending regular `AppendEntries` RPC heartbeats. When the cluster stops receiving heartbeats from the leader, it holds an election to elect the next leader. More details are described in the Raft paper, but the aforementioned details are the most relevant to this work.

This work is not the first to build DNS based on Raft, but it is unique for its focus on DDNS and work on write performance. Orbay and Fisher implemented Raft-based consensus among anycast-based DNS nameservers [12]. Although the Raft algorithm requires all requests, including reads, to go through the leader, Orbay and Fisher relaxed this restriction so followers could respond to read requests. While this adjustment risked returning stale data, it still respected DNS' requirements. Aariff et al. built on the prior work by comparing latency when DNS queries were sent to the leader compared to a load balancer routing the request to any node [4]. They found lower latency for requests through the leader, but the latency differed by under 5 milliseconds. We hypothesize that the latency difference might have been due to the extra network hop incurred with a load balancer. Bi et al. took Raft a step further by sharding DNS records between independent Raft clusters with consistent hashing [6]. These studies show Raft's viability for nameserver replication.

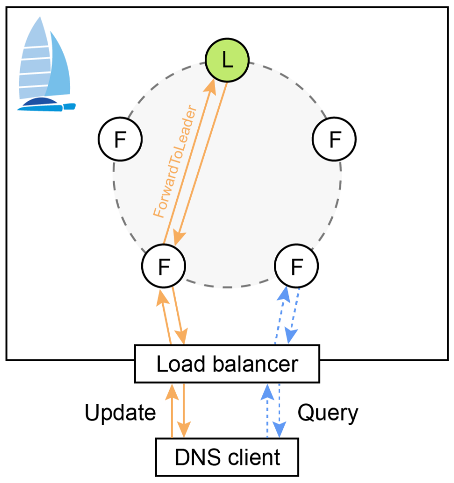


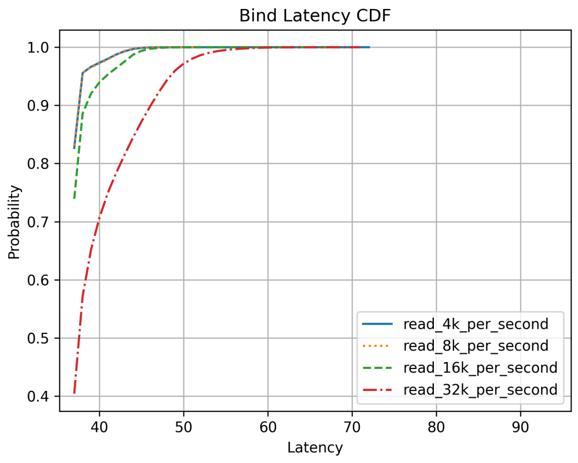
Figure 1: Catamaran system architecture.

3 DESIGN AND IMPLEMENTATION

Figure 1 shows Catamaran's architecture. Clients send requests to the IP address of a single external-facing load balancer, which distributes requests to the individual nodes. Our nameserver cluster nodes and load balancer were provisioned through Google Cloud Platform (GCP), so we used GCP's Maglev for load balancing in our experiments. The exact mechanism used should not matter as long as there is some method of balancing requests among the nodes, thus providing a single logical address to end-users, akin to anycast. Each node in the cluster effectively functions as both:

- a Raft node instance that keeps the replicated state, persists log entries to disk, and communicates with other nodes in the cluster using Raft RPCs, as described in [11].
- a DNS nameserver that can respond to both read and write requests from clients using common tools like `dig` and `nsupdate`, by passing these requests to the Raft side. As per the Raft algorithm, the nameserver waits until a DNS update has been safely replicated or times out before replying to the client.

We implemented Catamaran in Go. We used existing external libraries for running the nameserver logic and generating well-formed DNS messages and resource records [7], and for write-ahead logging/persistent storage [5]. During evaluation, we used a rate-limiting library [8] to send DNS requests at a specified maximum throughput. We implemented the basic version of Raft as outlined in [11] from scratch (with some extra modification), and used Go's built-in `net/rpc` library for sending and executing RPCs. The replicated state machine that the Raft layer applies commands to is a key-value store, mapping domain names (strings) to collections



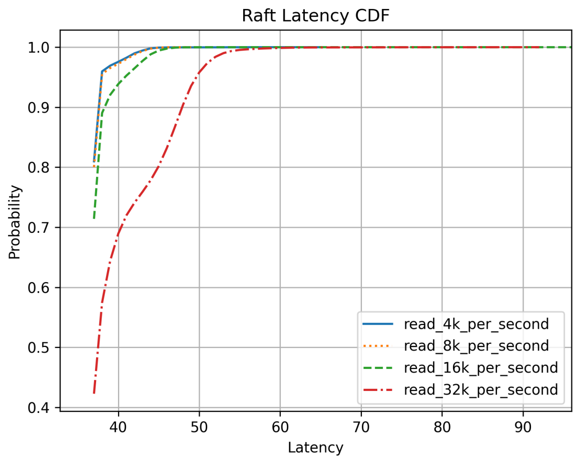


Figure 2: CDFs of read latency (ms) at different throughputs measured over three trials.

of DNS resource records. Each log entry contains a command to execute; they can either be `remove` commands with an associated key, or `update` commands to set a particular key and value (these are also used for creation); there is also a `blank` command that represents the *no-op commit* that new leaders make at the start of their terms. We chose not to implement the extra features of membership change support and log compaction due to the extra complexity involved.

We made three important changes to the Raft algorithm for Catamaran. First, we allow followers to respond to read requests. Although the original Raft paper stipulates that all reads must go through the leader to provide linearizability, linearizability is not a priority of DNS due to widespread local caching. Even though followers may return stale data, the lower read latency that this modification enables is a higher priority.

Second, in addition to the base Raft RPCs, we also add our own custom RPC, `ForwardToLeader`, for forwarding write requests from follower nodes to the leader node, since all writes must be done by the leader. The approach specified in [11] for followers to handle write requests is to reject the client's request and reply with the leader's direct IP, but this puts the burden on the client to retry the request with the leader's address. It does not provide a seamless experience for end-users and cannot integrate cleanly with standard, pre-existing tools like `dig`. We explicitly return an error only in the edge case where a write request is sent during an active election and reaches a node with a candidate status; the client will also receive an error if the RPC call times out, e.g. if the leader crashes before it is able to receive or respond to the write request.

Finally, we have the leader send batches of up to 10 log entries to its followers whenever it sends heartbeats. The original Raft paper suggests that multi-entry heartbeats can improve efficiency, and our experience confirms this suggestion. Batching helps our cluster handle higher write throughputs.

The DNS layer listens for requests from clients, translates them into requests to send to the Raft layer, and receives DNS resource records in return. These are packaged into the authority and answer sections of well-formed DNS replies and sent back to the client.

We also wrote a custom DDNS client in Go that can continuously monitor a host's IP address and send updates to a nameserver. This client supports issuing DNS requests at a specifiable maximum throughput, which we used for experimentation.

4 EVALUATION

We used GCP to evaluate Catamaran. For our cluster, we used five e2-standard-2 machines, each with 2 vCPUs and 8 GB of memory all within the same region and zone (us-west1-b). We provisioned a passthrough network load balancer to distribute clients' DNS requests across all of the five nodes. Another machine with the same configuration in a different region (us-south1-a) served as our testing node to issue DNS queries and updates to the load balancer, varying the maximum throughput via our custom DNS client.

4.1 Cost of Replication

To measure the cost of replication, we compared Catamaran to a simple BIND configuration on our cluster. We installed BIND 9 on three machines, where one was the primary server, and the other two were secondary servers. Reads were sent through the load balancer, but writes were sent directly to

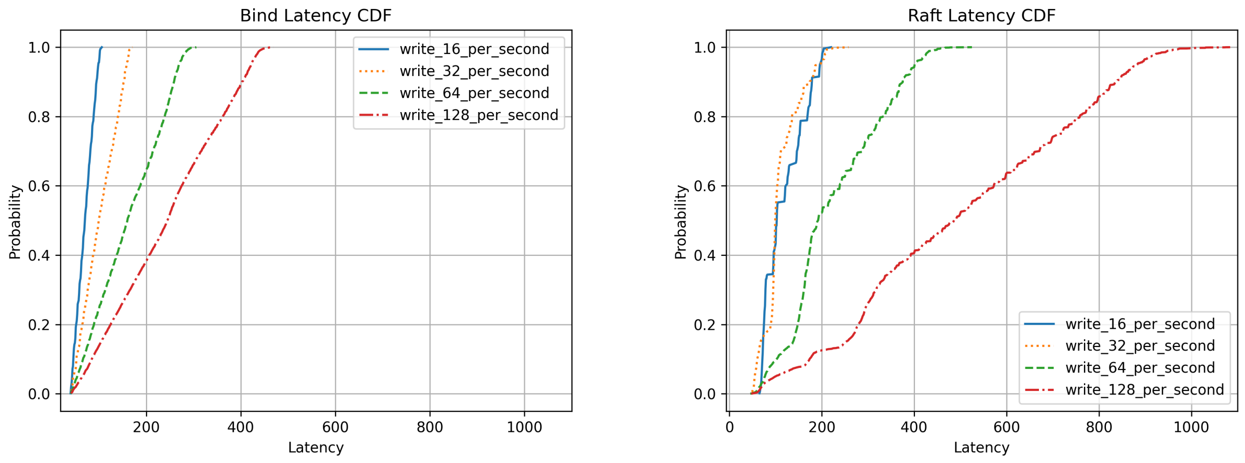


Figure 3: CDFs of write latency (ms) at different throughputs over three trials.

the primary server. We used our DNS client from the testing machine to issue queries and updates at a target rate. We issued read and write requests at the target throughputs specified in Table 1 and Table 2; these throughputs were approximate because the testing clients waited for a batch of requests to return before sending the next batch. This approach allowed us to better reflect real-world conditions, where the throughput jitters naturally, and evaluate higher throughputs without overloading the Raft and BIND clusters.

	Raft		Bind	
Max Throughput	Avg.	99.9th	Avg.	99.9th
4k	37	44	37	45
8k	37	44	37	44
16k	38	46	38	47
32k	40	60	40	59

Table 1: Read latency (ms) at different throughputs

	Raft		Bind	
Max Throughput	Avg.	99.9th	Avg.	99.9th
16	118	204	71	104
32	111	231	102	168
64	225	476	163	298
128	498	1030	243	459

Table 2: Write latency (ms) at different throughputs

Reads incurred a low cost of replication in Catamaran. The CDFs in Figure 2 are similar and Table 1 show similar average and 99.9th percentile latencies. Similar to BIND, where any server could answer a DNS query, any Raft node could respond to read requests.

In Figure 3, we can see that the cost of replication in terms of writes is non-negligible. For example, when making approximately 16 DNS updates per second, Raft had a 99.9th percentile latency of 204 ms, while BIND's was 104 ms. This difference became more apparent when writing more rapidly; when we made approximately 128 DNS updates per second, Raft's 99.9th percentile latency was 1030 ms, while BIND's was 459 ms. We observe this behavior because Raft processes every DNS update sequentially, and all DNS updates are forwarded to a leader with the ForwardToLeader RPC.

4.2 Fault Tolerance

To measure Catamaran's ability to tolerate node failures, we created a network partition to isolate a victim node from the rest of the cluster while our testing machine continuously sent either queries or updates to the cluster. The load balancer sends health checks every second, so it quickly detects the unhealthy victim node and directs incoming requests to the rest of the cluster.

This experiment occurred as follows. The cluster ran normally for 20 seconds before the victim node was partitioned. After 20 more seconds, the victim node was reintegrated into the cluster. The cluster ran for an additional 20 seconds. We measured both the query and update response throughput of the system during these events, and either the leader or a random follower was partitioned during our experiments.

We found that Catamaran was able to tolerate network partitions fairly well. When issuing queries and partitioning a follower, our system maintained a mostly consistent query at ~39k responses per second and write response throughput at ~130 responses per second (Figures 4 and 5, left panels).

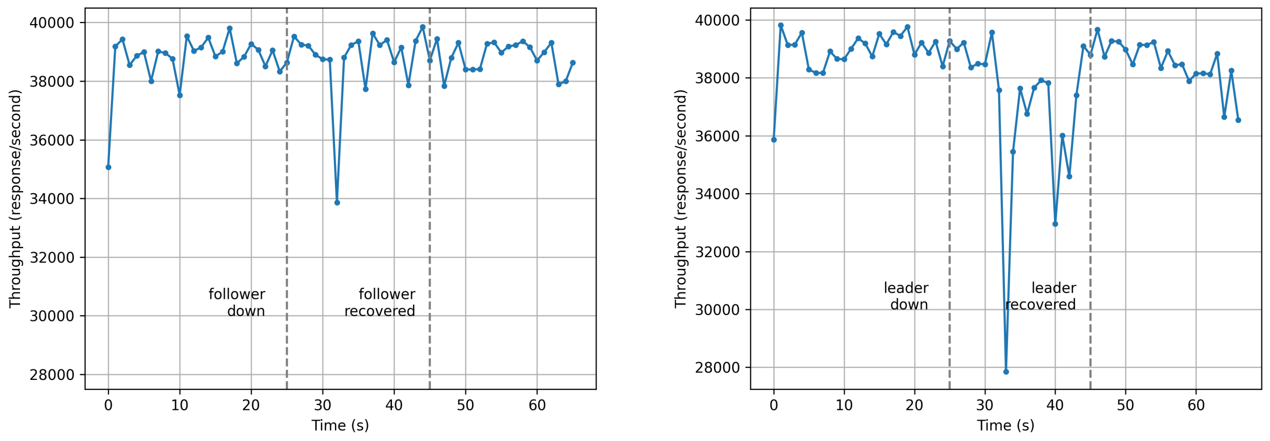


Figure 4: Fault tolerance with queries. Throughput is measured as responses per second. Labeled dashed lines indicate when the victim node was partitioned and added back to the cluster.

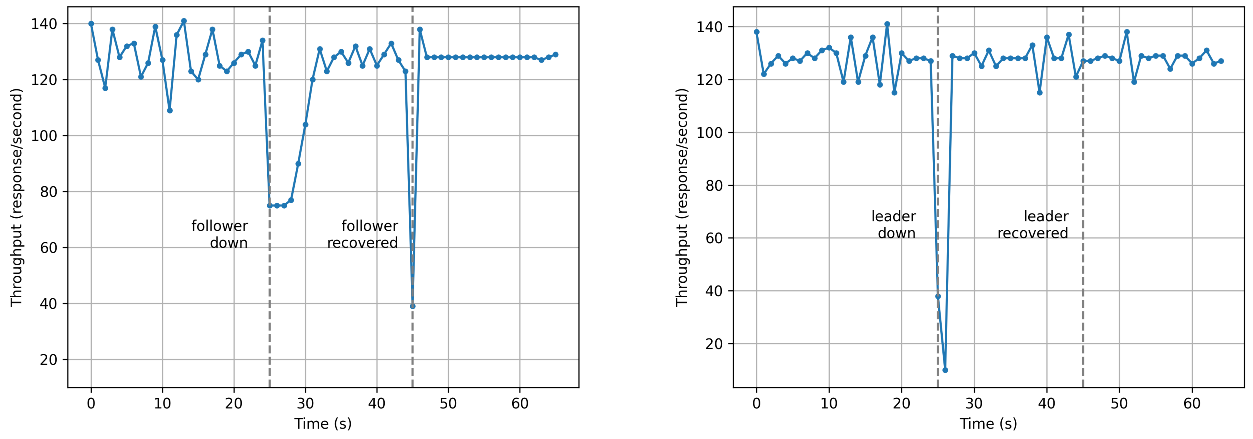


Figure 5: Fault tolerance with updates. As in Figure 4, throughput is measured as responses per second. Labeled dashed lines indicate when the victim node was partitioned and added back to the cluster.

When partitioning a leader from the cluster, we observed that the read throughput decreased by approximately 10%—~39k responses per second prior to the partition and ~35k responses per second during the partition (Figure 4, right panel). Interestingly, the write response throughput stayed largely consistent at around 130 responses per second, with a very short reduction in throughput (Figure 5, right panel).

In terms of writes, we found that partitioning a leader had a more adverse effect on throughput for a brief amount of time than when a follower was partitioned. For example, our system's write throughput dropped from 128 to 10 responses per second when partitioning a leader, while a follower partition caused the write throughput to drop from 128 to 75 responses per second. Likely, a leader dying caused an election, during which no progress could be made within the cluster, causing requests to queue up in the cluster.

The left panel of Figure 5 shows a massive, but short, drop in latency when the follower recovers. This behavior occurs because the follower floods the networks with outstanding RPCs when it recovers. Inspecting the follower node logs, we found that during its partition, it started many elections. The `RequestVote` RPC, which requests votes for leadership from peers, indefinitely retries in the absence of responses as per

[11]. Once the follower recovers, it has many outstanding `RequestVote` RPCs, thus preventing the cluster from making progress.

5 LIMITATIONS

This work serves both as a research and educational project for the team. One of the reasons that we worked on this project was to gain a deeper understanding of Raft. While we accomplished that objective, it also means that our Raft implementation has room for improvement. During evaluation, there were some transient failures that the team could not fully explain. Additionally, there may be some implementation issues regarding Catamaran's management of connections between cluster nodes. Specifically, we observed a multi-second period of low throughput caused by repeated elections when we entirely killed a Raft node, rather than partitioning it. We believe that these issues can be resolved with a bit more time, probably through improved synchronization (particularly by using read-write locks rather than mutexes), and a review of Raft's RPC retry requirements.

On the experimentation side, we recognize that BIND is not explicitly developed for DDNS. However, many DDNS services are proprietary services offered by a business, making it difficult to benchmark Catamaran against an actual DDNS service.

6 FUTURE WORK

With the results and limitations in mind, there are many paths for future work. First, Catamaran can become a richer service through implementing Raft's log compaction and membership change features. These features make Raft more efficient and give flexibility to domain administrators to scale clusters.

Second, it is worth exploring alternative consensus algorithms, perhaps those with more relaxed correctness constraints. Raft's requirement to immediately write log entries to disk for correctness' sake hampers its ability to support high write throughput, which is relevant to DDNS in certain scenarios. A less restrictive consensus algorithm could enable higher throughput and lower latency for reads and writes. For example, Kubernetes uses a Gossip protocol, or a peer-to-peer communication algorithm, to rapidly disseminate DNS records for Kubernetes APIs [3].

7 CONCLUSIONS

We presented Catamaran, a Raft-based replicated DNS system that provides low latency and high throughput for reads and moderate throughput and latency for writes. It provides comparable performance to BIND 9, while providing stronger fault tolerance. This system enables DDNS updates, so an Internet service can be accessed from a static hostname while operating on a dynamic IP address.

REFERENCES

[1] [n. d.]. BIND 9. https://bind9.net/
[2] [n. d.]. Cloudflare Learning Center: Dynamic DNS. https://www.cloudflare.com/learning/dns/glossary/dynamic-dns/
[3] [n. d.]. Gossip DNS — Kubernetes Operations. https://kops.sigs.k8s.io/gossip/
[4] Deen Aariff, Vishnu Narayana, and Zihao Li. [n. d.]. Evaluating Performance and Safety of Distributed DNS with RAFT. ([n. d.]).
[5] Josh Baker. 2023. Write-ahead log for Go. https://github.com/tidwall/wal
[6] Changyu Bi, Sean Decker, Kevin Qian, and Xinyi Yu. 2020. CRaft DNS: a robust and scalable DNS server on Raft. (2020).
[7] Miek Gieben. 2024. DNS library in Go. https://github.com/miekg/dns
[8] Google. 2022. Go Time — supplementary Go time packages. https://cs.opensource.google/go/x/time
[9] Ted Hardie. 2002. Distributing Authoritative Name Servers via Shared Unicast Addresses. RFC 3258. https://doi.org/10.17487/RFC3258
[10] Bryan Liston, Jeremy Cowan, and Efrain Fuentes. 2016. Building a Dynamic DNS for Route 53. https://aws.amazon.com/blogs/compute/building-a-dynamic-dns-for-route-53-using-cloudwatch-events-and-lambda/
[11] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm (Extended Version). https://raft.github.io/raft.pdf
[12] Emre Orbay and Gabbi Fisher. 2017. Distributed DNS Name server Backed by Raft. (2017).
[13] Michael A. Patton, Scott O. Bradner, Robert Elz, and Randy Bush. 1997. Selection and Operation of Secondary DNS Servers. RFC 2182. https://doi.org/10.17487/RFC2182
[14] Paul A. Vixie. 1996. A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY). RFC 1996. https://doi.org/10.17487/RFC1996
[15] Donghui Yang, Zhenyu Li, and Gareth Tyson. 2020. A Deep Dive into DNS Query Failures. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 507–514. https://www.usenix.org/conference/atc20/presentation/yang