Distributed DNS Name server Backed by Raft

Emre Orbay Gabbi Fisher

December 13, 2017

Abstract

We implement a asynchronous distributed DNS server backed by Raft. Our DNS server system supports large cluster sizes, high latency clusters, horizontal scaling, dynamic resource record management. We tested our system on a AWS EC2 and obtained throughput results that indicate fair performance.

I. Introduction

The Domain Name System (DNS) serves as a directory for the internet, mapping domain names to their appropriate IP addresses and mail servers. Name servers are a core component of DNS, and are responsible for storing resource records that provide information about a domain name's IP addresses, a domain's mail servers, and the topology of name server connections. DNS queries are performed in advance of connecting to web hosts, and must not introduce noticeable delays when a host connects to another. As such, DNS name servers must be highly available to answer queries with as little to zero latency as possible.

A popular design for highly available name servers is to maintain multiple name servers for the same set of domains. These name servers are distributed globally. Hosts access them through anycast, which routes a host to the closest physical name server instance[3]. Though these name servers are separated, they must maintain consistent resource records across multiple name servers. Thus, the application of a consensus algorithm is required to replicate the addition or removal of resource records.

This paper explores the utility of using Raft for resource record replication among a distributed system of name servers[5]. We built a distributed DNS name server using PySyncObj, a library that implements Raft to replicate Python data structures across multiple machines[6].

II. Prior Work

Distributed DNS name servers, accessed via anycast, are already deployed in practice. Research on replicated DNS services frequently examines the implementation of anycast[3]. RFC3258 proposed the concept of anycast DNS name servers[4]; despite the fact Leslie Lamport's original Paxos paper was published 13 years prior, the authors of RFC 3258 suggested that updates to name server zonefiles (lists of resource records) should be coordinated and executed in tandem by people at each node!

Research in this field often assesses the load-balancing capabilities of DNS server clusters reached by anycast.[1][2] Frequently, DNS name servers are described as replicated services and not replicated datastores of resource records. This follows a general trend in which papers about distributed DNS name servers focus on the implementation of anycast and its efficacy in load balancing, rather than the means of replicating resource records among service nodes.

Examining the use of Raft in a DNS name server cluster offers another lens for analyzing implementations of anycast DNS servers. Instead of analyzing anycast implementations

themselves, we assess the performance of an underlying consensus algorithm–the tool responsible for propagating resource records among anycast name servers.

i. Raft for Resource Record Replication

We elected to use Raft[5] for consensus because of its understandability in comparison to other consensus algorithms (namely, Paxos). Raft is able to provide the same fault-tolerance and performance as Paxos. The Raft consensus algorithm is designed to append entries sequentially to a log, which mirrors the periodic addition and deletion of new resource records well. Additionally, unlike Paxos, Raft boasts a set of mature open-source implementations in multiple programming languages.

Of particular importance for DNS name servers is a trade-off in favor of availability over safety. DNS servers must quickly respond to queries, even if they are missing newly added resource records or yielding stale records. This principle is present in Raft's replication approach: though new additions to a leading node's log are not propagated immediately to its followers, each Raft node is able to service client requests with information from its own log. While a follower's Raft log may initially be behind those of other logs, and consequently serve up stale information, DNS accepts that new resource records will not immediately propagate.

III. Implementation of DNS Clusters

i. Raft Replication

Raft is a consensus algorithm for managing a replicated log. Raft provides consistency and safety in the face of server failures and network partitions. In order to do this, Raft designs a system built of subcomponents such as leader election, managing membership changes, log replication etc.

Figure 1 illustrates how our system is designed. Each machine runs a DNS server and a Raft node on separate threads of execution. The Raft node maintains a log resource records which the server can query.

Raft allows only the leader to append entries to its replicated log. We use an abstraction on top of Raft to allow a network administrator to add and remove resource records from any node. The system would have been unusable If the administrator was required to figure out who the leader was to affect the resource records.

While our system is configurable to strictly use Raft as described in its paper, we think that most administrators would not want the name server to simply stop responding to queries in case of network or server failures. In our view, it is simply not acceptable for a DNS name server to block on queries if it can answer them, at least not without a back-up clusters. Thus, our system allows the network administrator to configure the DNS server to never block due to Raft, regardless of network conditions or server failure.

This asynchronous mode guarantees availability at the cost of safety. The DNS server simply accesses the Raft node's underlying datastore. This configuration, of course, throws out all safety guarantees that Raft provides. If the server is partitioned from the leader or during leader election and membership changes, it is possible for DNS server to (1) serve stale entries, (2) fail to respond with newly inserted entries and (3) respond to queries with NXDOMAIN even though the leader could have answered the query.

Raft makes no promises in the case Byzantine failures, and neither does our system in any configuration: nodes that are alive will continue to answer queries.

ii. Resource Record Data structure

The raft-replicated data structure used in our distributed DNS name server is functionally the same as a Python hash dictionary. The data structure is adapted from the Distributed-

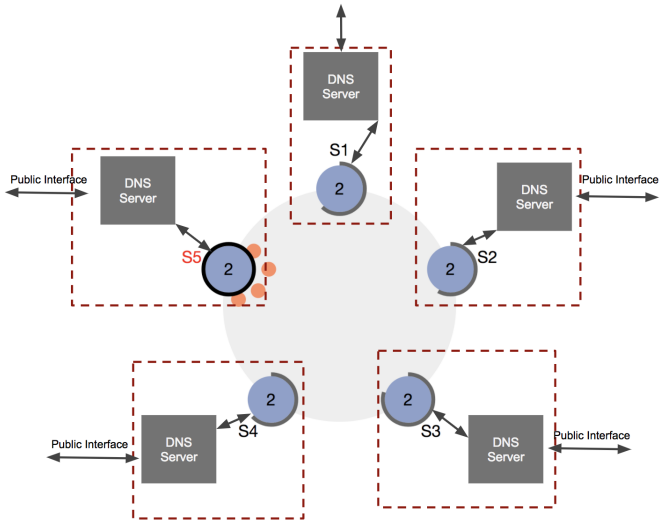

Figure 1: *Our DNS name server design. In this diagram, each physical machine is represented by the dashed boxes. Each machine runs a Raft node as well as a public DNS Server that can access the Raft replicated log. Each DNS server simply independently answers DNS queries by directly accessing the underlying storage inside the Raft node.*

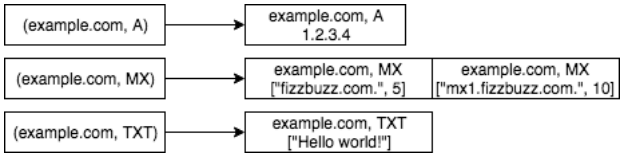

Figure 2: *Data structure for storing resource records*

Dict object provided by the PySyncObj library. DistributedDicts utilize Raft to replicate new key-value pairs added to the dictionary, in addition to propagating updates and removals of key-value pairs.

Resource records are stored in the replicated dictionary with the following scheme in Figure 2:

1. A key in the dictionary is a tuple of a resource name and resource type–for instance, (example.com, A)

2. A value in the dictionary is a hash set of resource records whose resource name and resource record match those in the value's corresponding key.

.

iii. DNS Name Server Cluster API

The name server API must support two functions: (1) constant-time, fast queries and (2) updates to the datastore's key-value pairs. Because DNS is lenient about the replication time for new resource records, writing updates to the datastore does not have to be as quick as responding to queries. We still seek to minimize latency in performing both actions.

Each node in the name server cluster can answer queries and commit writes to the distributed datastore.

Responding to Queries

A query (from a DNS resolver for directly from a client) consists of a question, which contains a domain name and record type. DNS queries are handled by the `query(question)` method

exposed by the name server API.

If the contacted name server node has the requested resource records, it takes the following steps:

1. Creates a key from the query's domain name and type,
2. Uses this key to index into the appropriate list of resource records in the distributed datastore
3. Serializes the list of resource records into the format necessary for the Answer section of a DNS query.

We use the Twisted Python library to serialize the list of resource records into DNS query format.

If the contacted name server node does not have the requested resource records – meaning the key formed from query name and type does not exist in the datastore – it simply returns an NXDOMAIN response.

We can see that this query call has a runtime complexity of $O(1)$, given that getting a value from the dictionary is a constant time lookup.

Adding Resource Records

A name server administrator can add resource records by calling `addRecord(resource_record)`.

In this API call:

1. A key is created from the new resource record's name and type
2. The key is used to index into the set of existing resource records of that name and type (if they key does not exist in the datastore yet, a new key-value pair of the key and an empty set is added)
3. If the new record doesn't already exist in this set, it is added to the set.

If the name server that is also the Raft leader node receives the `addRecord()` call, it replicates the new resource record through the Raft log replication process. If a follower name server receives the `addRecord()` call, it redirects the client to talk to the leader node. While this redirect introduces latency, the propagation of a newly added record is not necessarily immediate.

Excepting latency introduced by Raft, the runtime complexity of this call is $O(1)$. Checking for the preexistence of the new record is an $O(1)$ set membership check. Adding to the set is another constant time process and writing the updated set to the datastore is also $O(1)$.

Removing Resource Records

Similarly, an administrator can remove a resource record by calling `removeRecord(resource_record)`. This call is similar to `addRecord()`, but it instead removes a resource record from a set instead of adding it. It follows that resource record removal is also an $O(1)$ process.

IV. Evaluation

i. Loopback Experiment

To evaluate performance, we benchmarked the number of `dig` queries the name server cluster could answer in a second. We ran varying counts of name server nodes as individual processes on the same machine with a 2-core GHz Intel Core i5 CPU and 8 GB of RAM. We queried all nodes from the same machine and recorded the number of queries answered per second. The processes were connected via the loopback interface. Then, we ran as many `dig` processes as the machine can handle.

The results are shown in Figure 3. It appears that the throughput of the system goes up as the number of clusters and thus availability increases.

ii. AWS Experiments

We also launched three separate tests on AWS EC2. All tests involved a cluster of 3 EC2 nodes connected in a virtual private cloud. `iperf3` reported that network throughput between the cluster nodes as well as the query

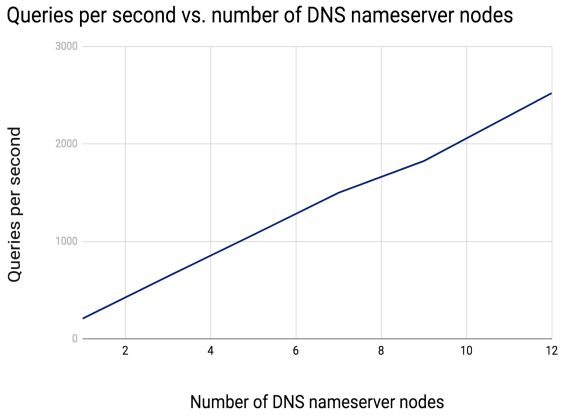

Figure 3: *Results of the loopback test. We ran 6 tests with different cluster sizes from 1 node through 12 nodes. It appears that the number of queries scales linearly on the loopback interface. Here the bottleneck is speed of the system.*

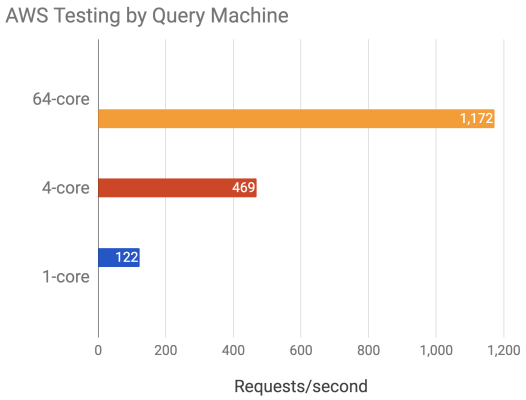

Figure 4: *Results of the AWS tests. When only a single core machine was querying, the cluster returned 122 DNS responses per second vs. 469 responses/second when 4 separate cores are querying. This shows that the throughput of our system is CPU-bound, and could feasibly scale as long as someone pays for more cores.*

node was 1Gbps and network latency was sub-millisecond. For the first test (shown in blue in Figure 4), the cluster was made up of t2.micro nodes, each with a 1-core vCPU. For the second and third tests, the cluster was made up of t2.xlarge nodes, each with 4-core vCPUs.

Discussion

Figure 4 shows the results of the three AWS tests. Y-axis indicates the number of cores on the EC2 instance issuing the queries through `dig`. This machine was not in the cluster, but was inside the same subnet as the cluster. It appears from these tests that the bottleneck is the core count in the system. While 4-cores on the query machine received 4x as many replies as the 1-core machine, the 64-core machine only received 3x as many replies as the 4-core. This can be attributed to the fact that there were only 12 cores in the cluster, hence the cluster compute power was the limit.

The throughput was calculated end-to-end, i.e. time from when the first request left to when the last answer was received. We suspect that our testing scheme–through performing `dig`–introduces latency because sending query questions and flushing query answers introduces overhead on the name serverâĂŹs query lookup. Another testing scheme with less overhead may be necessary for more accurate query/second metrics.

iii. Fault Tolerance

Though Raft will fail to propagate updates to the resource record datastore if over half of its follower nodes have failed, each DNS name server node is capable of answering queries with its current (though possibly incomplete) commit log. Given that DNS servers do not have to guarantee complete safety, returning NXDOMAINS for unpropagated new resource records is acceptable.

As a result, this DNS name server cluster can answer queries even if only one node is up; however, if a majority of servers are down, it cannot guarantee the propagation of new or removed resource records.

V. Future Work

To continue our exploration of using Raft to replicate resource records among name servers, further tests are needed to explore potential

bottlenecks in Raft's ability to propagate resource records. Some of these tests include:

- Testing performance with a large number of querying nodes to identify a plateau in how many queries a DNS name server can answer in a second. Real DNS servers answer over 100,000 queries a second, and our implementation must undergo further load testing to compare its performance to that of industry DNS name servers.
- Testing queries/second when name server nodes are geographically distant.
- Testing propagation speed among name servers following the insertion or removal of a resource record.
- Serverless computing offers a novel testing platform: we can launch AWS Lambda Functions to flood our cluster with requests.

In addition to the testing, we've found that we are bottlenecked by compute power. Implementing all of our code in Python certainly cuts down own what we can accomplish per core. A more efficient Raft implementation in a compiled language and a faster DNS server would yield much better throughput.

VI. Conclusion

This project utilized Raft to build a distributed system of DNS name servers. We found that Raft potentially suits the replication of resource records because DNS name servers guarantee availability over safety. Each node in our distributed DNS name server was able to answer hundreds of queries per second, and test of resource record replication demonstrated that Raft correctly updated resource records. Additionally, our DNS name server exhibited fault tolerance, and was able to continue answering DNS queries with just one functioning node. Nonetheless, a majority of nodes must be functional to guarantee resource record propagation. More rigorous testing of a Raft-backed name server is necessary to justify Raft's usage in resource record propagation.

Our work is open-source and may be reviewed at `https://github.com/gabbifish/distributed-dns`

References

[1] Hitesh Ballani, Paul Francis, and Sylvia Ratnasamy. A measurement-based deployment proposal for ip anycast. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 231–244. ACM, 2006.

[2] Peter Boothe and Randy Bush. Dns anycast stability. *19th APNIC,'05*, 2005.

[3] X. Fan, J. Heidemann, and R. Govindan. Evaluating anycast in the domain name system. In *2013 Proceedings IEEE INFOCOM*, pages 1681–1689, April 2013.

[4] Ted Hardie. Distributing Authoritative Name Servers via Shared Unicast Addresses. RFC 3258, RFC Editor, April 2002.

[5] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[6] Filipp Ozinov. Pysyncobj. `https://github.com/bakwc/PySyncObj`, 2017.