CRaft DNS: a robust and scalable DNS server on Raft

Changyu Bi, Sean Decker, Kevin Qian, Xinyi Yu
(Ordered alphabetically by last name)

June 1, 2020

0 Abstract

We present CRaft DNS, a replicated DNS name server architecture with high availability, fault tolerance, and transparent storage scalability. The key to CRaft's design is the use of consistent hashing to shard zone records among independent clusters of Raft nodes: consistent hashing servers direct read and write requests to these storage cluster members, which then process requests and replicate them among peers through the Raft protocol. We show that CRaft can be scaled out through the addition of extra clusters or of new nodes to existing clusters, tolerates non-Byzantine failures, and is useful for real world DNS query workloads. We validated our design assumptions with concrete benchmarks on AWS EC2.

1 Introduction

The Domain Name System (DNS) represents a fundamental pillar of the internet, translating human readable domain names with IP addresses, among other information, that can be used to locate computer services and devices on the Internet. Because of this vital role, DNS server outages can cause web pages and email to be unreachable, rendering these internet services unusable. Furthermore, slow DNS lookup slows down the entire experience of using any web service, regardless of the number of load balancers or web servers.

The most common provider of DNS services are ISPs, but because DNS is only a peripheral, non-billable part of their service, ISPs often do not prioritize the availability, and ISP-based DNS is subject to regular outages and they often take a long time to recover. There are third party DNS service providers such as Dyn that try to address some of the issues [1]. However, their solutions are proprietary, and such nature means that their mitigation strategies will not become available in time for other providers to follow when a new problem is uncovered.

In this paper we present CRaft DNS, a transparently scalable, highly available, fault-tolerant DNS server architecture with open-sourced implementation. CRaft DNS achieves this through the use of sharded storage of DNS records in Raft clusters with consistent hashing servers directing traffic to these clusters. These consistent hashing servers use responses from cluster members as implicit heartbeat messages in order to verify their liveness. With these properties, CRaft DNS could serve a variety of use cases. In particular, it could be valuable in an IoT network: with increasing numbers of smart devices, a large number of DNS records will be needed to address each device instance. IoT devices also tend to have a higher failure rate, meaning a moderate amount of record updates are necessary during replacement. We expect that CRaft DNS could tackle these two requirements.

To verify the viability of CRaft DNS, we benchmark both throughput and fault-tolerance, and demonstrate that CRaft DNS functions well also on a real world workload. We show through our throughput benchmark results that our use of clusters for shards allows our max throughput to scale close to linearly with the number of nodes in our clusters, meaning that our system can be scaled out instead of scaled up. Our fault tolerance benchmark results shows that our hash server does provide more higher availability by allowing our service to recover from faulty nodes. Finally, through our investigation of real world DNS traffic, we show that our service can also be useful in a real world setting.

2 Related work

Our system is designed and implemented based on several algorithms and methods. We use the Raft algorithm to achieve replication consensus on multiple storage nodes that form a cluster, and use consistent hashing to shard resource records among different clusters.

Raft consensus algorithm. Raft [2] is an algorithm designed to achieve consensus of a replicated state machine, providing high availability and fault tolerant properties in distributed systems. It is a protocol easier to understand than alternatives such as Paxos, and has its key elements clearly illustrated: leader election, log replication, membership changes, etc. We adopt the Raft algorithm in our system to gain consensus between multiple storage nodes that store DNS records and maintain clusters.

Etcd Raft library. Etcd is an open-sourced, distributed and reliable key-value store adopted by popular frameworks like Kubernetes as a backend. It implements the Raft algorithm as a library with stability and full feature. In 2016, this Raft library is claimed to be the most widely used one in production [3]. We utilize this library as the Raft algorithm implementation for our system.

Consistent hashing. Consistent hashing [4] is a distributed hashing mechanism that is widely used for distributed caching. It partitions data according to the hash value of the key to some places on a ring, and lets each machine take charge of data on the small portion of the ring. The algorithm provides scalability for storage and flexibility for cluster configuration changes. We use it to partition DNS records among different Raft clusters.

3 Design

Our design is mainly around a single conceptual DNS nameserver. It behaves just like a normal DNS nameserver that can be installed into the DNS resolution flow, which potentially maintains its own zone, while also allowing records pointing to other subzones in the hierarchy. It supports both iterative and recursive resolution.

The conceptual name server consists of 2 major types of components: varying number of *storage clusters*, each storing a partition of all local records, and a set of *consistent hashing servers* at which external queries and updates are served.

Each storage cluster takes the responsibility to store a partition of resource records. We run Raft protocol inside of each cluster, ensuring records are reliably replicated and consistent among cluster members, while tolerating member failures. Except for during resharding and migration, there are no communications between the clusters, and the presence of each is unknown to peer clusters.

The consistent hashing servers receive and respond to incoming DNS queries and update requests. Each of them are present as NS and glue records in the zone above. These servers are aware of the Raft clusters, and use consistent hashing to assign key ranges to each of them, thus allowing scalability on a large amount of records. Depending on whether there are frequent changes to the storage clusters configuration, these hashing servers can operate independently, or forming a cluster themselves. Since they maintain very little state (only cluster information), a crashed consistent hashing server can recover almost instantly.

Each member node inside of a storage cluster maintains 3 separate interfaces: the Raft protocol interface, DNS UDP interface, and a DNS update interface (currently implemented as HTTP requests). The Raft protocol interface handles messages about term leader elections and log replication. The DNS UDP interface sits on the

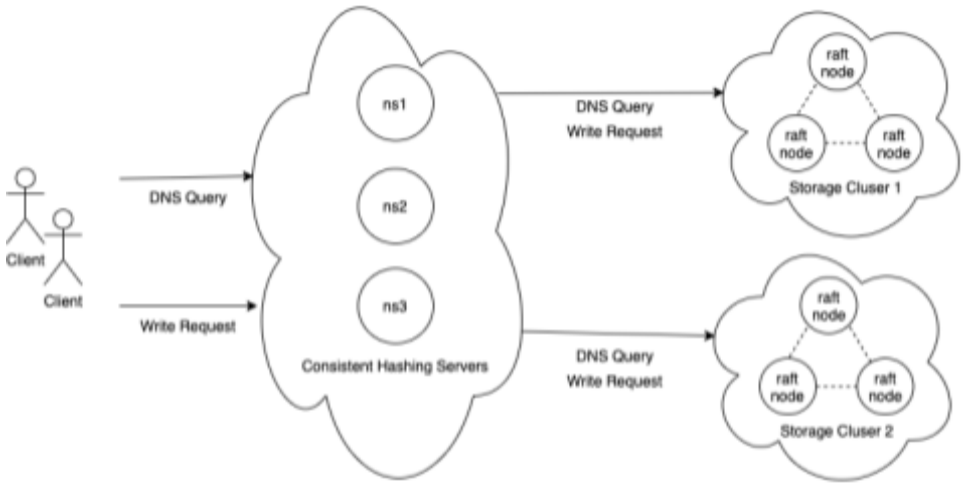

Figure 1: High level overview of the design. A set of public-facing consistent hashing servers receive external requests, and forwards them to the corresponding partition of storage cluster, where Raft is running among the members

conventional port 53 for DNS nameservers, and is expected to receive DNS query messages as defined by RFC1035 [5]. These messages are expected to be read only and thus can be handled locally by the recipient node. In the case of recursive queries, we might optionally maintain a cache, for which we could either choose to have this cache private or also replicated, the latter requiring us to potentially issue writes inside of the cluster. The DNS update interface receives requests for adding and deleting records. Using an HTTP REST API, the current implementation supports adding a record in the form of submitting a text representation of the resource record, and deleting a list of records for a name of a certain type.

Instead of explicitly maintaining a master zone file, the DNS records are usually stored in-memory in member nodes for efficient lookup (the records can be easily exported as zone files). In the case where we want to limit total memory usage, an optional simple least-recently used (LRU) paging policy is implemented, which allows records to be organized as pages and can be paged out to the filesystem when number of pages present in memory exceeds a user defined limit (some metadata, such as last applied commit log index, are tracked per page to allow reapplying commits idempotently during crash recovery). Write ahead logs are maintained as dictated by the Raft protocol, and periodic snapshots are taken to allow faster recovery on node failure. On read-only queries, the node can directly respond using its local records if available; on record addition and deletion requests, the requests are serialized and forwarded to the leader and the commit will be propagated to all peers; actual update to local records is done only after the request is committed.

Consistent hashing servers also maintain a similar DNS UDP interface and an update interface, but do not save any resource records themselves. Using the cluster information and consistent hashing, these servers decide which cluster should they forward the request to and obtain the response from. It might seem redundant for storage clusters to implement normal DNS query support even when they are not facing the public. This is however intentional: it allows the storage clusters themselves to function as a set of fully-working nameservers, such that we can completely drop these consistent hashing servers when a single cluster is able to satisfy storage requirements.

On receiving an external DNS UDP message from the interface, the consistent hashing server computes the partitions to which the request should be routed to, based on the domain name of the question. This might involve more than one cluster in case of multiple DNS questions with different domain names (note that such queries are rare in practice). It then decides on a list of preferred member nodes in each relevant cluster, and issues batched requests to their corresponding interface in parallel. The preferred lists are ordered based on the last known timestamp where the node is considered alive, with randomness introduced to avoid hot nodes. The first nodes of these lists are forwarded the queries, and only on response timeout are subsequent nodes tried. If the queried cluster member node does not respond in time, the server considers the node temporarily unavailable, and updates its alive timestamp to some moment in the future, effectively making the node no longer preferred until the timestamp is passed, assuming that it will take some time for the node to recover. If some of the designated partitions replies to a DNS question without the expected record, it is likely that the record does not exist, while it is also possible that a wildcard record covering the name is present instead. In that case, the unanswered question will be broadcasted to all clusters. While this is obviously not the most efficient strategy, we expect queries that do not yield a result (or yield a wildcard result) to be rare compared to our major use cases (for some specific use case, we can also choose to disallow wildcard records instead). After receiving the replies from clusters, the consistent hashing server combines the answers and replies to the external client, in a way such that the presence of storage clusters is completely transparent.

4 Implementation

We use Go to implement CRaft DNS. We prefer Go over scripting languages or platforms such as Node.js, where cooperative multitasking precludes CPU parallelism [6], and Python, where Global Interpreter Lock prevents true parallelism. We also prefer it to C/C++ due to its simplicity, built-in memory management to allow faster development, and effective light-weighted green thread implementation (goroutines). We also considered Rust due to its elegant design and strong safety guarantees, but eventually did not do so since it is a harder language for everyone on the team to pick up quickly.

We mainly use 3 open source libraries: `miekg/dns` for DNS message serialization and deserialization (we implemented basic DNS iterative and recursive resolution ourselves since it is not provided by the library), `buraksezer/consistent` for basic consistent hashing, and Etcd for Raft protocol. As discussed above, Etcd is a well-written distributed and reliable key-value store that is adopted in popular systems such as Kubernetes. However, instead of directly using Etcd, we

choose to rip out only its Raft protocol implementation, since we realized that while DNS records are feasible to store as pure key-value pairs by proper serialization, such implementation would introduce extra performance cost due to constant parsing. Also, by directly using the Raft APIs, we have a better control of its snapshotting utilities and write-ahead log format, therefore allowing us to tweak their configuration parameters when deemed necessary.

Two independent binaries are generated: `dns_server` for storage cluster nodes, and `hash_server` for consistent hashing servers. As discussed before, their independence allows partial benefit from our design without using all components. `dns_server` can also operate under the optional paged mode that takes advantage of disk storage through the `--page` flag.

5 Benchmark

To validate our design decisions and show the practical usability of our DNS service, we explain in this section the benchmarks conducted on our service that measure throughput, fault-tolerance, and the latency against real-world DNS traffic.

5.1 Scalability of a Raft cluster

We first validate our design choice of using a Raft cluster to hold DNS data and serve DNS queries to improve read throughput. We measure the maximum throughput of a single Raft cluster with a varying number of nodes. Each Raft node is an instance of an Amazon EC2 t2.medium node with 2 vCPU and 4GB of RAM. We use 3 t2.2xlarge nodes (8vCPU) as our client nodes to saturate the servers with DNS queries. We first populate the cluster with 1 million DNS records, which takes more than half of the memory (each node is using about 2.4GB of memory). The network delay between any two nodes is submillisecond. The network bandwidth of each node is 1Gb/second. In this experiment, each client issues as many requests (uniformly at random) as possible to obtain the max throughput. The experiment result is in Figure 2.

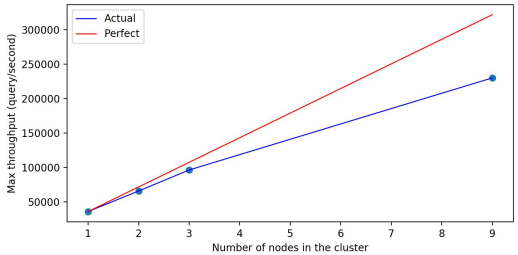

Figure 2. Results of the scalability experiment. The perfect line represents perfect scalability with slope 1. Our client nodes became network bottlenecked in the case of the 9-node cluster.

At the baseline with a one-node cluster, the max throughput is about 35 thousands queries/second. The two-node cluster and the three-node cluster give about 1.8x and 2.7x baseline throughput respectively, which are close to the perfect scalability (the red line in Figure 2). With a nine-node cluster, our client nodes became network-bottlenecked and the throughput is about 6.5x the baseline. This result matches our expectation that the max read throughput should increase almost linearly as we add nodes to the Raft cluster, since in our design read operations do not go through the Raft protocol and each node is able to answer queries by checking its own data.

5.2 Throughput with hash servers

We then experiment and measure how consistent hashing servers affect the performance. We use the result of a three-node cluster from Section 5.1 as our baseline and consider the following settings: one hash server + one three-node cluster, three hash servers + one three-node cluster, and three hash servers + two three-node clusters. The machine types for hash servers and cluster nodes are all t2.medium. Note that ideally there should be a consensus protocol between hash servers to ensure a consistent view of configurations. We did not implement this in our benchmark due to time concern, but the effect on throughput should be minimal since the configurations are fixed during each experiment. We use the same three client nodes as in Section 5.1 to measure the max read throughput in each setting. The results are in Table 1.

	3-node cluster	1 hs + one 3-node cluster	3 hs + one 3-node cluster	3 hs + two 3-node cluster
Max TP (query/s)	96125	17731 (5.4 : 1)	46504 (2:1)	53344 (1.8:1)

Table 1. Max read throughput of different settings.The ratios are compared with baseline (the column with 3-node cluster). The bottleneck is the CPU at hash servers.

If we compare the results in the column of "3-node cluster" and the column of "3 hash servers + one 3-node cluster", we see a 2x downgrade in performance compared to the baseline. So introducing hash servers to direct DNS queries reduces about 50% of the max throughput. This can be attributed to the extra processing logic at the hash servers, e.g., computing hash functions and parsing query questions and results. In addition, since hash servers serve as a "proxy" to direct DNS queries and answers, the latency is almost doubled in our setting where all nodes are in the same AWS availability zone. To retain the same throughput with this increased latency, the hash servers

need to handle a larger number of client threads which may also introduce some CPU overhead.
Note that this latency overhead should be minimal in practical settings where the latency is dominated by the message delay between clients and server nodes.

This throughput reduction matches our prediction, and we expect that this overhead can be reduced by optimizing the implementation of the hash servers (e.g., use more efficient data structures), caching DNS query results, and properly batching DNS queries. It is worth noting that by introducing hash servers, we can have multiple clusters of Raft nodes and this enables our service to store more records in memory for fast service. In fact, in the setting of the last column of Table 1, we stored 2 million resource records while this is not possible to be stored in memory in a single cluster for all other settings in Table 1. We expect a similar result under the paged version of storage cluster nodes.

5.3 Fault tolerance and availability

This part tests the fault tolerance of our system. For write operations, we rely on Raft features: it can tolerate failure even when a minority of nodes in a cluster fail, while for read requests, we apply local reads so that as long as the queried node is alive, read operation can succeed. We further add a hash server layer which enables hiding more failures, mainly due to the hash server's strategy of node-querying-priority inside one Raft cluster that it tends to query an alive node.

We decide to do all experiments on pure read requests, for this is the most prevalent use case of DNS service. In the first experiment, there is no hash server running in the system. Multiple client threads on a single machine directly send DNS read requests to one random server in the Raft cluster, and that storage server replies back to the client. In the second experiment, one hash server is set up. There is only one Raft cluster, therefore the hash server just does the forwarding of requests and replies. The client threads send all DNS read requests to the hash server, and receive responses from it. We use the following configurations for this part: one client machine of t2.micro type, one Raft cluster composed of 3 storage machines of t2.micro type and one hash server of t2.micro type (if used).

In both experiments, we do a time - throughput measurement, as shown in figure 3. In the beginning, every server is alive. At 30 second, we kill a server in the Raft cluster. At 60 second, we kill another server in the Raft cluster. At 90 second, the first server is restarted and finally, at 120 second the second server is restarted.

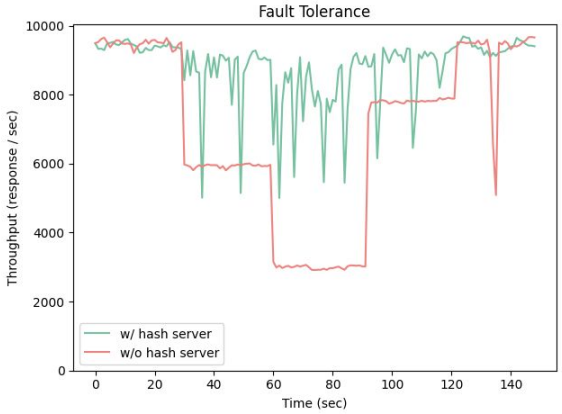

Figure 3. Results of the fault tolerance and availability test.

The results appear as expected. The red line (first experiment) represents the local read properties of the basic Raft cluster: when killing one of three servers, the systems can still handle read requests and the throughput drops to 2/3. When two of three servers are killed, it cannot handle write requests (not shown here), we still allow read requests to happen locally, and the throughput drops to 1/3. After restarting the two servers, the throughput goes back to the original value.

The green line is for the experiment with a hash server. It starts with the same throughput as the red line, but generally it has higher throughput than the basic Raft cluster line when several servers are dead. That is due to the strategy adopted by the hash server: after failing querying some nodes inside the hashed Raft cluster, it will prioritize other nodes in that cluster during a period of timeout. Therefore, one or two of three servers' death doesn't have too much impact on the decrease of throughput: the hash server will try to query the alive server. The dive of the figure is related to the timeout mechanism of the hash server when it occasionally retries a previously dead node to see if it becomes reachable again. While the disruption is negligible in practice, it would be an interesting experiment to eliminate recurring dives by sending separate explicit heartbeat messages before retry if reviving a node takes a long time.

The experiments show that our system possesses high fault tolerance properties, and even better availability due to the existence of hash servers.

5.4 Experiment with Real World Traffic

In order to verify the practicality of CRaft DNS, we also need to verify that it can handle real world queries generated in a probable client network. We must ensure that CRaft DNS can respond to DNS queries generated in a real world setting as quickly as other high availability DNS nameservers. This experiment will help to validate our design by ensuring that real world DNS traffic doesn't have a certain pattern that is hard for CRaft DNS to deal with. Furthermore, there may be a difference in the interaction between CRaft DNS and a home network device versus CRaft DNS and a server running in a datacenter.

	mean time	standard dev
Google	6.94	5.82
Cloudflare	4.80	1.54
CRaft DNS	6.35	3.38

Table 2. Table shows the mean response time for 2,433 DNS queries. The experiment was run 18 times.

To run this experiment, we first needed to gather a large sample of DNS traffic. This was done using Wireshark. Through wireshark, we were able to gather a sample of 6 hours of DNS queries from various devices connected to a home wireless LAN. 2 computers and 3 smartphones were connected to Wifi at some point during this sampling. This monitoring allows us to collect 2,433 DNS queries.

After gathering these queries, we then populated a nameserver running CRaft DNS with the responses to this real world DNS traffic. This server was running on 2 clusters, each with 3 nodes each, with each node being an EC2 t2.micro instance.

To test the performance of CRaft DNS on this real world traffic, we then selected Google's 8.8.8.8 nameserver and Cloudflare's 1.1.1.1 nameserver to compare against. From a home network, all 2,433 were sent in 35 separate threads to each nameserver and the time it took for each to respond to all requests was recorded. This experiment was iterated 18 times and the results are in [table 2].

We find that CRaft DNS has comparable response time to these two other larger DNS nameservers, though the standard deviation of response time is quite high. We hypothesize that the more important factor driving response time are artifacts of the network as opposed to the nameserver itself. This result still verifies that our DNS name server is useful in the wild and that even on real world DNS traffic it performs comparably well as a real world DNS nameserver.

6 Future Work

Since our team members have few prior experience with distributed systems, we spent a long time understanding the protocols, learning Go, seeking libraries, and figuring out how to use Docker and AWS for local and remote deployment and testing. Therefore there are some points we realized during development yet unable to evaluate, given the time limit. We are listing some of these below as future work ideas if we were given more time to explore.

We realized early on the necessity of storage partitioning to avoid full replication of the record database. We went to use consistent hashing with virtual nodes to achieve such. However, as pointed out by systems such as Coral [7] and Dynamo [8], there are other alternatives to build a distributed hash table such as Kademlia, CARP, or the simple equal sized partition tables. More careful experiments with different strategies might help to reveal interesting policies the public-facing servers can apply.

Consistent hashing servers in our current implementation treats each incoming request as independent and forwards queries independently to storage clusters. However, it is also possible to batch questions from different outside queries into a single DNS message for forwarding, thus reducing total message exchange count and byte amount (merging multiple UDP packets/DNS messages into one implies only 1 header is needed) between components. We were hesitant to implement such a strategy, since batching means we need to determine the length of an appropriate time frame during which we will wait for more packets to come in before forwarding to internal clusters. A fixed time frame might either be too long to have terrible latency, or too short to even see a second packet coming in, and synchronization costs might be too high between originally independent tasks executed. However it is still an interesting idea to evaluate with more concrete experiments.

We implemented a simple migration strategy for the in-memory version of the storage cluster nodes: temporarily withholding write requests in consistent hashing servers, copying migrated records from existing clusters to the new cluster, reenabling write requests and triggering migrated records cleanups inside each cluster using a dedicated garbage collecting thread. This allows clusters to continue serving with zero read downtime while migrating in the background. However, it might be

possible to find another strategy that can allow write requests during migration. We also did not have time to implement the strategy for the paged version of storage cluster nodes. However, one simple design involves similar steps as above, with some tweaks: the hashing servers enter read-only mode, temporarily stop forwarding write requests to clusters (to avoid the complexity of concurrent updates), and signal the existing storage clusters to flush out all pages. The new cluster can now retrieve these pages (possibly using some network file system) and move entries that hashes to the key range controlled by itself, and leave markers about which entries are migrated, and can thus be dropped by their old owners when paged in again. When complete, the consistent hashing servers are notified and can finally switch to use the new membership and re-enable write request forwarding. This ensures that the existing records continue to be available for DNS queries, and keep re-sharding completely transparent to the user.

Raft is introduced to ensure strong global consistency guarantees for updates. However, as revealed by Dynamo and COPS, there are weaker models that can still fit the use case of DNS. For example, causal+ [9] might also be effective: since DNS has, for example, CNAME types for name aliasing, it has the risk of unsatisfied dependency if updates are reordered by network, therefore causal+ can tackle these potential issues, and we won't need to pay the extra cost for, e.g., ensuring the temporal order among updates on independent domain name keys.

7 Conclusion

As an extensively used service that affects almost all internet services, DNS should be highly available and performant. DNS services should also be scalable to meet the need of the growing number of hosting requests, while still being fault-tolerant. The report by Dyn pointed out that the common DNS service providers, ISPs, failed to meet such requirements [1].

In this paper, we introduced CRaft DNS, a DNS name server design that provides high availability, fault-tolerant and scalability. CRaft uses the Raft protocol in each DNS server cluster to provide fault-tolerance and adopts consistent hashing servers on top of DNS server clusters to provide transparent storage scalability. This two-level architecture also ensures the high availability of our DNS service. We showed that CRaft DNS can scale horizontally through the addition of nodes and clusters through benchmarks performed on a test name server running on AWS while still maintaining fault tolerance. Through these tests we showed that CRaft DNS would be a viable architecture for a real world DNS name server. In particular, a CRaft DNS name server would be valuable to systems which require more writes because of common device failures and high scalability such as in IoT use cases.

8 References

[1] "The Case Against Free ISP DNS," https://dyn.com/wp-content/uploads/2013/06/CaseAgainstFreeISPDNS.pdf (June 2013).

[2] Diego Ongaro, John Ousterhout, "In Search of an Understandable Consensus Algorithm," *USENIX Annual Technical Conference* (2014).

[3] Etcd authors, README of Etcd Raft library, https://github.com/etcd-io/etcd/tree/master/raft (Retrieved June 1, 2020)

[4] David Karger et al., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (May 1997).

[5] P. Mockapetris, "Domain Names - Implementation and Specification," Network Working Group Request for Comments 1035 https://tools.ietf.org/html/rfc1035 (November 1987).

[6] Atul Adya et al., "Cooperative Task Management without Manual Stack Management," *Proceedings of the 2002 Usenix Annual Technical Conference* (June 2002).

[7] Michael J. Freedman, Eric Freudenthal, David Mazières, "Democratizing content publication with Coral," *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (March 2004).

[8] Giuseppe DeCandia et al., "Dynamo: Amazon's Highly Available Key-value Store," *ACM SIGOPS Operating Systems Review* (October 2007).

[9] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS," *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (October 2011).