Evaluating Performance and Safety of Distributed DNS with RAFT

By
Deen Aariff, Vishnu Narayana, Zihao Li

Abstract

The RAFT consensus algorithm is a tool that can be used to implement highly available distributed DNS services much more easily than the Paxos algorithm, which is the industry standard. Despite the ease of implementation, not much research has been done into the difference of performance and safety when different metrics of the algorithm differ. In this paper, we will test implementations of the RAFT consensus algorithm with varying tweaks in order to test previously made assumptions of the algorithm.

Introduction

A. Objective

Our Objective is to Implement a distributed DNS service using the RAFT Consensus Algorithm where we evaluate the performance and safety of the system based upon multiple criteria.

1. Querying only the leader versus any node in the system
2. Different percentages of nodes alive in the system
3. Synchronous vs asynchronous methods of querying the underlying data store in RAFT nodes

B. What is the Problem

Distributed DNS are at their core distributed data stores, and therefore consensus algorithms can be utilized to implement them. One solution is the RAFT consensus algorithm, which was formulated as an easier to implement solution over Paxos, the long held academic standard for distributed consensus.

However, implementations of RAFT can differ in various ways. The problem of distributed DNS poses unique problems, much of which vary in its different use cases. The rise of Microservices, and the use of DNS for service discovery adds more complication to this issue. Therefore, further experimentation is required to assess whether certain metrics will differ, specifically performance and safety (the correctness of the records returned from the DNS) when operating it on the RAFT consensus algorithm.

C. Why this is a project related to this class

Domain Name Services (DNS) are a key component of the architecture of the internet, as virtually all users of the internet must consult a Domain Name Service to resolve hostnames in IP Addresses. Therefore, DNS is also integral in meeting the need to resolve hostnames of services that are made available on the Cloud. However, the importance of DNS extends beyond the scope of how Cloud Computing Infrastructures are accessed and to how they function as well.

For example, the trend towards microservices in the software industry has created an environment in which DNS is often used for service discovery (the ability to discover services whose IP address is volatile). A parallel phenomenon is the importance placed on the DNS service used in the systems implemented to orchestrate microservices, such as Apache Mesos, Kubernetes, and the HashiCorp Stack. This use of DNS to query custom names for services, means that the role of DNS adopts many of the traits of systems running in the cloud, including scalability and fault tolerance. Distributed DNS comes into the picture because it provides the fault tolerance and robustness necessary for other systems to rely upon it. Therefore, a project that addresses the Distributed DNS is tightly coupled to trends in Cloud Computing that rely on functionality such as service discovery.

Such trends are expected to grow and become more developed. Therefore, an experiment addressing Distributed DNS may produce results that are very influential and applicable to modern Cloud Computing infrastructures.

D. Existing Approaches

The proposal to use RAFT in Distributed DNS environments is not novel and other have produced academic literature addressing its use. However, many take the assumptions that hold in Globally Distributed DNS Service and extend them to all distributed DNS Services.

For example, in the paper “Distributed DNS Name Server backed by RAFT”, the assumption is made that within a globally distributed dns availability must be emphasized over safety, therefore querying follower nodes in a distributed DNS is an acceptable practice. This is despite follower nodes being potentially behind leader in log updates. While this may hold true when using RAFT in a globally distributed DNS where records do not change often, the opposite is often true in DNS used for service discovery. In such microservice environments, service names are often being mapped to new IP addresses quite frequently and as a result querying follower nodes may result in being returned stale data. While an incorrect DNS record, may not be a byzantine failure in a cloud environment, it may produce unwanted performance delays that may propagate and effect the performance of the entire system.

The RAFT consensus algorithm, as well as its relative Paxos, propose theoretical proposals for distributed consensus. The RAFT consensus algorithm holds that as long as a majority of nodes are alive in a cluster, the consensus algorithm can operate as expected. However, this constraint may not extend to performance as well. In order to test whether the performance of a distributed DNS using RAFT will remain consistent during failures, further testing should be done to ensure we can rely upon RAFT in the case of Highly Available Distributed DNS.

Two types of network communication that are popular are asynchronous and synchronous communication, also commonly referred to as non-blocking vs blocking-io. Asynchronous communication often offers benefits in scalability, and when used in a single threaded environment such as that of Node.js, it can offer benefits in scenarios of high network traffic. However, it becomes impractical in the case of performing heavy server-side requests. Blocking IO, is typically augmented by multiple threads to increase performance. It is a popular method of network communication, and is implemented in many web-based server side applications. Since, DNS record lookup is an O(1) implementation given the underlying data structure is a HashTable, the gain that may be achieved through a synchronous network communication is diminished. This offers the possibility of evaluating the performance of asynchronous (non-block IO) network communication in Distributed DNS Scenarios.

E. Our Approach

We will elect to evaluate the performance and safety of querying only the leader in the distributed DNS versus querying all nodes in the system. This method carries inherent performance versus safety drawbacks, as querying a single node has the potential to undercut availability. However, querying followers provides the potential to return false records. We believe that our data will demonstrate that a configuration of querying a leader versus querying only the follower, should be emphasized in certain scenarios and requirements of users. This will counter the belief, that in the realm of distributed DNS backed by RAFT, it is always acceptable to query all nodes in the system.

The RAFT consensus algorithm proposes the availability of the system in the face of a non-majority of nodes dying. However, this does not extend to the performance, and perhaps safety of the system during these non-byzantine failures. Therefore, we will elect to see how Distributed DNS performs and maintains safety in the face of non-majority failure on nodes. Seeing that these metrics are important for distributed DNS on both a macro and micro scale, we believe this aspect of our experiment to be important.

Blocking and nonblocking IO offer distinct advantages. By looking at performance metrics of blocking vs non-blocking IO for querying values from RAFT nodes side by side we will gain insight into how RAFT performs under this change in network communication.

F. Scope of Investigation

Our experiment will attempt to observe the performance and safety along the following criterias.

1. Querying only the leader versus any node in the system
2. Different percentages of nodes alive in the system
3. Synchronous vs asynchronous network connections to query data from the RAFT cluster

The methods of how we will perform our analysis as well as the proposed implementation will be mentioned in subsequent parts of this paper.

Hypothesis

Among the three subproblems in which we are evaluating the performance and safety of RAFT, we hold three separate hypothesis.

Firstly, strictly querying a leader in a RAFT based distributed DNS will lead to greater safety but poorer performance. By electing to query follower nodes, the result will be greater performance but poorer safety. This hypothesis stems from RAFT's assertion that follower nodes may not be up to date with the leader node. Therefore, we may access old information for a given key-value pair from a follower node.

In regard to the second subproblem, a non-majority failure among the nodes in a RAFT cluster will lead to decreased performance as well as lower safety. The likelihood of this increases significantly when a leader dies and new leader must be elected.

Lastly, non-blocking communication across nodes will serve to increase the performance metric but not affect safety. Based upon the architecture of RAFT cluster, this is theorized to be due to the increased performance of the leader being able to handle a high volume of requests.

Theoretical Bases and Literature Review

A. Definition of the problem

In many papers on RAFT, certain aspects of the algorithm are accepted solely based on theory and assumption, rather than being tested. This makes the adoption of the RAFT algorithm for uses other than DNS networks difficult due to the lack of concrete information.

B. Theoretical background of the problem

Not much technical background is needed to understand this problem- the algorithm itself is proven to be quite efficient in its own right, but the assumptions lie in how the algorithm itself is used rather than the efficacy of RAFT compared to similar algorithms. We looked through several papers about the RAFT algorithm to see if the assumptions stated in our objective were answered. We found that certain aspects of the assumptions were scrutinized (eg. various speeds of RAFT implementations using asynchronous networks were measured in relation to each other and the number of nodes each network had), but they weren't compared to an implementation that used a synchronous implementation. In other words, most benchmarks of the algorithm that we found did a poor job of comparing tested factors to a proper control setup, and the ones that did have proper controls did not test against the factors presented in our objective.

C. Related research to solve the problem

Three papers were of relevance to us when exploring the general direction of research into the RAFT algorithm. Those papers were “Distributed DNS Name server Backed by Raft,” “Dynamo: Amazon’s Highly Available Key-value Store,” and “In Search of an Understandable Consensus Algorithm.” These papers gave us the understanding of how RAFT is generally analyzed (to other algorithms and with scalability rather than to different variations of RAFT), as well as a potential other use for the algorithm beyond DNS networks.

D. Advantage/disadvantage of those research

The related research we gathered allows us to understand the motivation of many other RAFT researchers and what they wished to accomplish in their papers. This understanding was then used to formulate the list of testing criteria listed in the objective. Our objective was formulated based on what factors and comparisons were not made in other research papers that we found to be the most important for more deeply understanding the intricacies of the RAFT algorithm. The disadvantage to us brought about by those papers stems from the fact that our goal is to expand the knowledge base of the RAFT algorithm in an entirely new direction, one where there aren't other papers which we can use to compare our results to. In such a case, the papers themselves don't create a disadvantage, rather our goal creates such a disadvantage that would exist no matter what related research was explored.

E. Your solution to solve this problem

The proposed solution to the problem is multi part. First, we will identify certain aspects of the RAFT algorithm that have little to no research performed on them. Next, we will design an implementation of systems that reflect use and lack of use of said aspects. Then, we shall decide on how to test and judge the the pairs of systems based on relevant criteria (such as performance and safety). Finally, we shall evaluate the results of the tests and come to a conclusion about the difference between the use and disuse of each aspect.

F. Where your solution different from others

Our solution is based on the scientific method, in which we find aspects to test, perform research on the aspects, formulate a hypothesis, perform tests, and formulate a conclusion. The main difference between our research and others' is that we ask different questions, specifically ones that don't have solid answers yet.

G. Why your solution is better

Our solution is better in the sense that it seeks to answer questions that don't have solid answers yet.

Methodology

A. Input

We have two inputs.
One is the new pair of domain name and IP address which will be sent to the Distributed DNS System by a Helper. All of them should be different. The other input is the query request from the client. The client will send the request to the server periodically and wait for the response.

B.Solution

We will measure two metrics on three cases. The metrics are the performance and safety. Performance means the how long the client can get the response from the server. The safety means the percentage of the correct IP address the client get from the server.

For performance, the client sends query requests to the server periodically and waits for the response. A timestamp will be attached on each request. When the client gets the response, it will compute the response time. We will use the average of the m intervals to represent the performance.

$$\text{performance} = \frac{\Sigma\text{ (response time of request i)}}{\text{total network requests}}$$

For the safety, the client sends query request to the server periodically and it will also upload new pairs of domain name and IP address at the same time. After the client get 1000 response, it will compute the percentage of the correct IP address. We use this percentage to represent the safety.

$$\text{safety} = \frac{\Sigma\text{ (1 if correct, 0 if incorrect)}}{\text{total network requests}}$$

The three cases to test these metrics on are:

1. querying the leader node and one of other nodes in the Distributed DNS serve
2. killing non-majority nodes in the distributed DNS servers
3. implementing the communication between Distributed DNS servers in both synchronous way and synchronous way.

When querying the leader node and one of the other node in the clusters, we have two clients. One will send the query request directly to the leader and the other one will send the query request to the load balancer first. The load balancer will randomly pick up a node in among the Distributed DNS servers. We will then compare both the performance and safety.

In the second case, we will randomly shut down non-majority numbers of nodes and measure the performance and the safety. We will also evaluate cases in which the leader was shut down as opposed to a follower node.

When implementing the communication between DNS servers using asynchronous vs synchronous communication, we will implement the communication in both ways and compare the performance and the safety between them.

C. Implementation Specifics

For our implementation of RAFT-based DNS, we built atop an existing Open-source solution for a Distributed Key-Store implemented using RAFT called Weave. It is available at the following link.
Our modifications included, fixing various errors in the leader-election conditions that the implementation used. Additionally, we added the ability to query data from the cluster using socket communication in both a blocking and non-blocking (java NIO library) fashion.

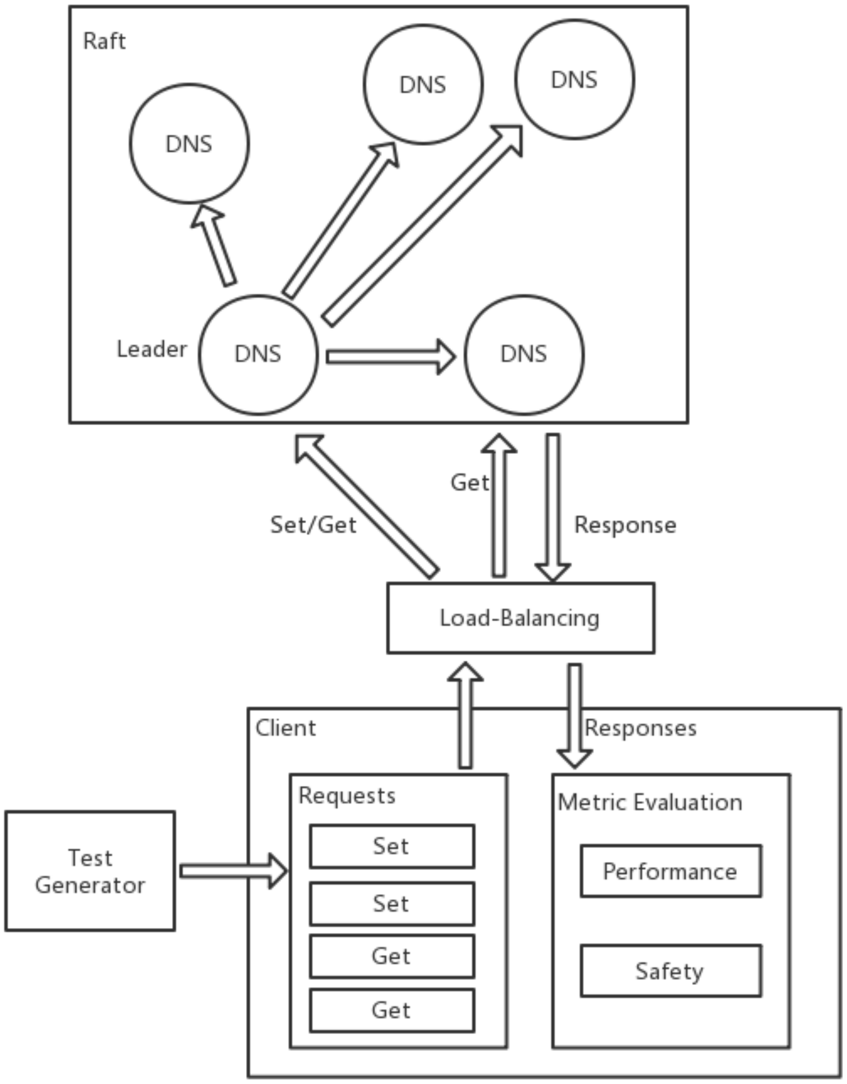
The new Classes created were:

ClientController/SocketClientController.java
ClientController/SyncSocketClientController.java
ClientController/AsyncSocketClientController.java

We also added the capacity for a node to "die". We do this by simulating the state and volatile variables that would be set when a actual failure would occur in a RAFT cluster.

Implementation

Our project will be divided into four parts. They are test generator, client, load-balancer and DNS name servers running Raft, as it shows in the picture below:



First of all, the load balancer runs firstly. It gets the IP address of the machine it runs on and the available port on which it can establish the socket connection. Then it opens a socket and wait for requests from the DNS name Server or the client.

Then DNS name servers connect to the load balancer and do the leader election process. Everytime the DNS name servers elect a new leader, the leader will send a message to the load balancer. The format of the message show below. Then load balancer compares the election time and sets the new leader's attribute on it and waits for the requests from the client.

New leader:

```
{
   “cmd” : “leader”,
   “leader_IP”: “ip”,
   “leader_port”: port
   “election-time”: int
}
```

The requests sent by client are read from the cmds.txt. They are generated by the Test Generator. For each requests, the client will establish a socket connection with the load balancer and send the request to it. After that, the client will wait for the response from the load balancer.

Test Generator can generate two kinds of tests. The first one is the basic test. There will be 100 set requests and 100 get requests after them. The second one is the weighted latest test, which will mix the set requests and get requests randomly. Because the leader needs time to propagate the new value to the followers, the pair which is the most recently set will have safety problem. We used a weighted selection technique, which let the most recently set pair has the much higher probability to do the get request.

When the load balancer receives the requests from the client, it will parse the request first and get the command. The command can be set, get and kill. Set command means setting a new pair of domain name and IP address in the DNS servers. Get command means getting IP address of the specific domain name. Kill command is used to kill the specific DNS name server. This is used for one of the test cases.The load balancer will redirect the request to the specific server.
The format of the commands shows below:

Setting:
```
{
      “id”       : the unique identifier for the cmd
      “cmd”  : “set”,
      “var”     :  the domain name to be set
      “val”      :  the value of the domain name
      “msgid”: index of the message for the current test. Used for debugging
}
```

Getting:
```
{
      “id”        : the unique identifier for the cmd
       “cmd”   : “get”,
       “var”     :  the domain name whose IP address is wanted
      “leader” : “True” / “False”
      “Msgid” : index of the message for the current test. Used for debugging
}
```

Kill Node: Wait 800ms after sending command before sending next one
```
{
        “Id” : the unique identifier for the cmd.
        “cmd”: “kill”,
        “val”: “any” or “follower”
        “msgid”: index of the message for the current test. Used for debugging
}
```

After the DNS name server get the request, it will sends response to the load balancer. The load balancer will redirect the response to the client. The format of the response is

```
Response:
{
    “cmd” : “get” or “set”,
    “id”     : the unique identifier for the cmd. Should match the id of the query this is a response to
    “valid” : True - no problem, False - error occurred,
    “var”   : the variable queried
    “val”    : the value returned by query
}
```

After the Client get the response, it can do the metric evaluation. It will compute the performance and the safety metric, and show them out.

Data Analysis and Discussion

A. Output Generation

The client, upon receiving a response from the server on a sent get or set query, will record the particular result, as well as the time it took to receive the result from when it made the corresponding request for it. Once all responses from a sequence of queries are received by the client, the average response time for the get queries, the average response time for the set queries, and the rate of accuracy of the get queries is printed. Below are tables that record these results for different variations of how the queries are sent over 5 separate trials which use different sets of queries.

Performance get	Leader Only	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Blocking (Leader)	Yes	1.297	1.175	1.328	1.173	1.009
Blocking (Any)	No	3.404	4.382	3.551	3.733	4.694
Non-Blocking (Leader)	Yes	1.412	1.152	1.389	1.157	1.256
Non-Blocking (Any)	No	5.196	3.64	3.599	3.703	4.304
Kill Nodes (Leader)	Yes	1.359	1.501	1.435	1.5	1.183
Kill Nodes (Any)	No	1.714	1.113	2.655	1.271	3.173

Table 1: This table shows the average response time for "get" queries in milliseconds issued by the client using different variations of how the load balancer interacts with the RAFT cluster.

Performance set	Leader Only	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Blocking	Yes	1.326	1.349	1.414	1.271	1.108
Blocking	No	1.176	1.181	1.621	1.425	1.481
Non-Blocking	Yes	1.416	1.285	1.586	1.376	1.373
Non-Blocking	No	1.673	1.572	1.133	1.622	1.57
Kill Nodes	Yes	2.533	1.775	1.729	1.559	1.304
Kill Nodes	No	1.121	2.751	1.362	3.054	2.544

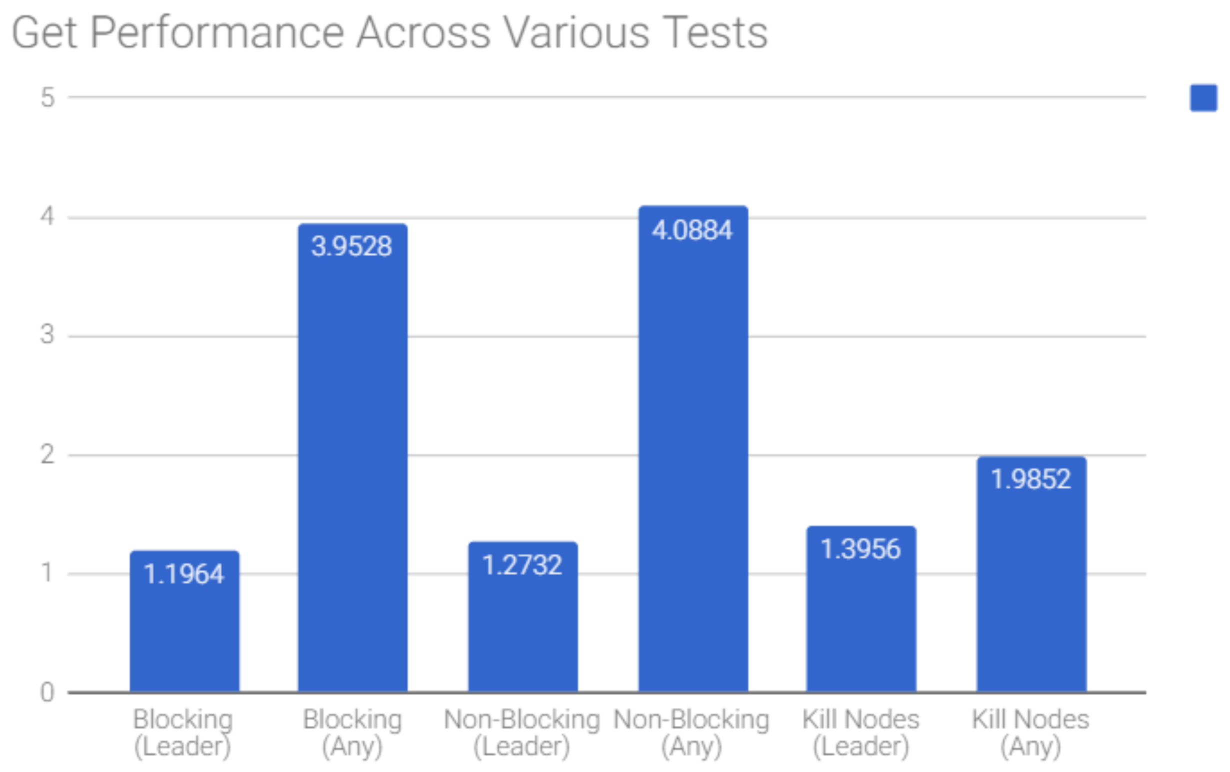
Table 2: This table shows the average response time for "set" queries in milliseconds issued by the client using different variations of how the load balancer interacts with the RAFT cluster.

Safety	Leader Only	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Blocking	Yes	0.952	0.8974	0.96	0.909	0.822
Blocking	No	0.5714	0.7948	0.78	0.836	0.711
Non-Blocking	Yes	1	0.8974	0.94	0.909	0.977
Non-Blocking	No	0.5714	0.7948	0.58	0.818	0.822
Kill Nodes	Yes	0.314	0.282	0.34	0.182	0.267
Kill Nodes	No	0.93	0.897	0.94	0.964	0.956

Table 3: This table shows the accuracy of “get” queries issued by the client using different variations of how the load balancer interacts with the RAFT cluster. The accuracy is in the form of a decimal, which equals (number of correct responses / number of “get” queries made). 1 represents 100% accuracy, and 0 would represent 0% accuracy.

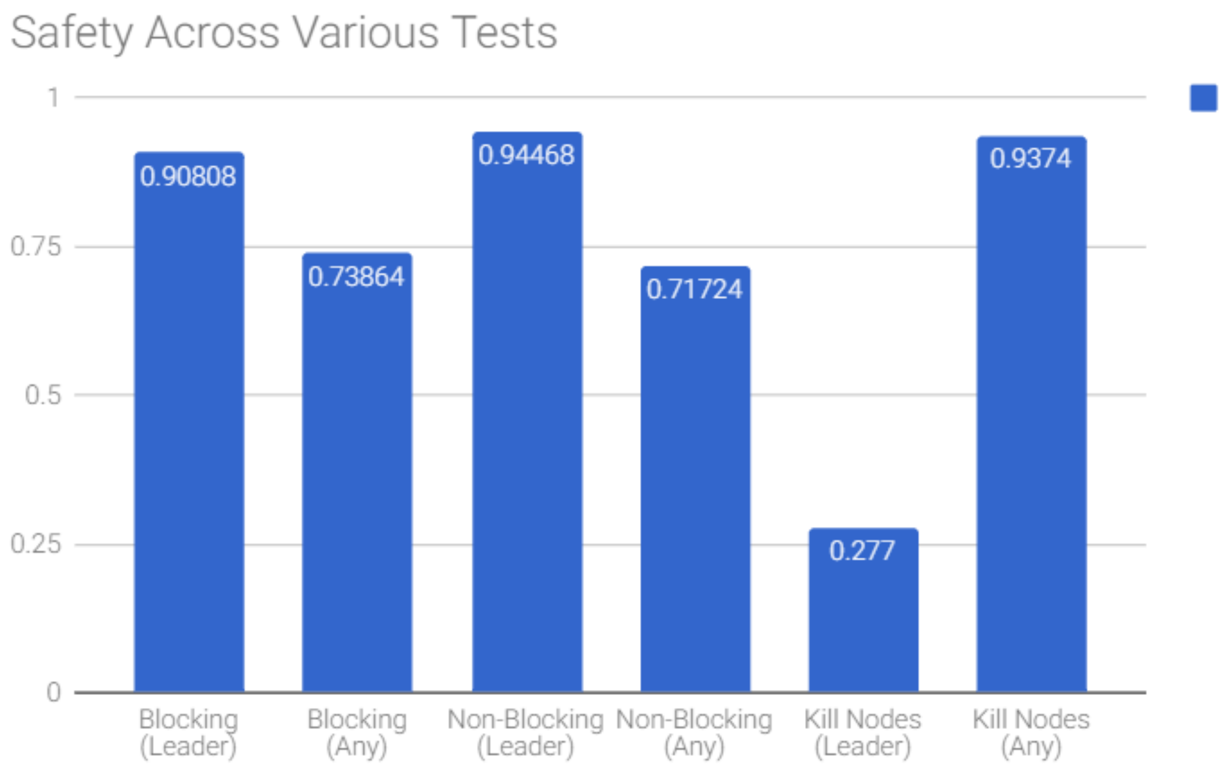
B. Output Analysis

There are several key observations that can be made by analyzing the data. Our hypothesis involved predicting how different configurations of and interactions with the RAFT network would affect the safety and accuracy of queries made.



Graph 1: This graph shows the average response times in milliseconds for "get" queries made by the client using different configurations of the RAFT network.

Looking at the response times for different network configurations, the following trends can be observed. First, querying the leader is faster on average than querying any node on the network for each of the network configurations tested. Second, the non-blocking configuration of the network on average has a slightly slower response time than the blocking configuration, though this is small enough to not consider. Third, the configuration of querying any node while killing nodes yields an average much lower than that of the other network configurations that also query any node.



Graph 2: This graph shows the average rate of accuracy for "get" queries made by the client using different configurations of the RAFT network.

The observations from the above graph are as follows. First, querying the leader for all network configurations except for the one where nodes are killed results in a greater accuracy than querying any node in the network. Second, queries made using the blocking configuration of the network has a slightly smaller accuracy for querying only the leader than querying any node, though this change is extremely small. Third, when killing nodes, the accuracy is extremely small when querying the leader versus querying any node.

C. Comparison to Hypothesis

Our hypothesis made several predictions. The first prediction was that querying only the leader would result in greater safety but worse performance than querying any node. From our results, excluding the test where we kill nodes, the safety is in fact higher when querying only the leader versus any node. Unlike our hypothesis, querying only the leader resulted in a better performance than querying any node.

The second prediction made in our hypothesis was that non-majority failure among the nodes in the cluster will result in decreased performance and lower safety. The non-majority failure was simulated by the "Kill Nodes" configuration of the queries sent. The safety of failure was much lower, with a much lower accuracy in tests where non-majority failure was simulated. The performance was a little different, though. When querying only the leader, performance increased very little, but when querying any node, performance was much greater than when there was no node failure.

The third prediction states that non-blocking communication across nodes will increase performance but not affect safety. Our results show that for when there isn't any node failure, non-blocking communication increases performance by very little. They also show that queries made only to the leader increase safety by very little but queries made to any node decrease safety by very little.

D. Abnormal Case Explanation

The most abnormal data we received was the performance of querying any node on the network when we simulate non-majority node failure. For some reason, it has a much higher performance when querying any node on the network when there is no failure. One would think that the increase in accuracy could have been tradeoff with the wrong or an invalid response being returned by the network, but it hosts a very high safety rate with its performance. Perhaps many queries could have been made during a leader re-election, which could result in the node sending back its stored value without consulting other nodes on the network. If the node had the correct value stored in it previously, then it would return the proper response, resulting in a high accuracy along with the performance.

The second abnormal result was how querying the leader of the RAFT network when simulating non-majority node failure results in a much lower accuracy than when querying the leaders without failure. When a leader node is killed, a new leader must be elected. Perhaps leaders could've been killed before they properly propagated the values from set queries, leading to new leaders not being able to give fully accurate results when receiving get queries, as some nodes would have older, improper values.

E. Discussion

Overall from our results, we were able to show the performance and safety of the RAFT algorithm given different configurations of the network. The trends of RAFT as shown by the data give several insights on potential problem areas in how the algorithm could see failure. First, when node failure occurs, there could be massive losses in accuracy. This could be caused by the time it takes to discover a new leader, in which case, the network should pause any incoming queries and only process them once the leader is elected and the network recovers from the killed node. Second, querying the leader has much better performance and safety than querying any node, so applications that simply use nodes as a means of storing data while only needing to query from a single location would be the best match for the RAFT algorithm. An example of this is in microservices which need to record key-value pairs with a large margin for safety and performance.

There were several changes to our data collection that we could have implemented to achieve more insight on the algorithm's results. First, we could have used larger trial sizes and more trials. This is a very basic change and these increases could have helped the data reach a more representative average, but for the purposes of our introductory study as well as due to time and resource constraints, what we did is enough for now. Second, a slightly more in-depth analysis of accuracy could have been implemented. When the cluster responds to a get query, the client records the response as incorrect if the value returned was incorrect or if the response was invalid, usually caused by a server error. Some of the unusual data could potentially be explained by errors occuring, which would make sense if a leader wasn't elected by the time a leader-only query was made.

Conclusion

When looking at the results of our metric evaluation, we can that some of our initial hypotheses were met while others did not hold up to our metrics.

For example when when purely querying the elade we expected a greater level of safety, however worse performance due to the majority of the stress being placed on the leader node. However, we actually noticed slightly better performance on average when querying from purely the leader as opposed to all other nodes. This may have to do with the fact, that followers are under heavy stress from processing heartbeat responses. Nevertheless, it is an interesting observation that holds importance in the realm of distributed DNS used in a non-global environment. However, some of our observations were correct. We suspected that in the scenario of a node failure, that we would experience a decline in performance. Our results partially confirmed, this as a leader death had a significant result on safety (the worst out of any of our test groups). Thus this demonstrates the significance of having a leader-failure in a RAFT-based Distributed DNS when safety is critical, such as in private network environment for rapidly changing IP (microservices). Our hypothesis that the effects of blocking vs nonblocking IO would be negligible were correct as well.

Throughout the process of the experiment, we noticed the vulnerability that RAFT based cluster could have when experiencing extremely high volumes of subsequent get and set queries. Our ability to test this was done by our random indexing that was weighted by the most-recently set query. Furthermore, our willingness to explore the assumptions that safety is not a key metric in Distributed DNS, holds false in RAFt-based Distributed DNS that operates in local networks with high throughput. Therefore, it is worth further exploring the applications of different type of consensus in the various environments Distributed DNS might be deployed in.

The results of this study were interesting, and it would be worth evaluating the similar performance metrics across another consensus algorithm such as PAXOS or one of its variants like Zookeeper. Furthermore, the importance of safety and performance in globally vs locally distributed DNS is highly worth investigating. Therefore, we were pleased by the ability of our experiment to underlie the importance of challenging certain assumptions and paving the way for further research in the subject.

Bibliography

DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." *ACM SIGOPS operating systems review*. Vol. 41. No. 6. ACM, 2007.

Ongaro, Diego, and John K. Ousterhout. "In search of an understandable consensus algorithm." *USENIX Annual Technical Conference*. 2014.

Orbay, Emre, and Gabbi Fisher. "Distributed DNS Name server Backed by Raft." (2017).

Appendices

A. Documentation

Overview

This Distributed Application evaluates a RAFT-based DNS cluster along the metrics of performance and safety.The application consists of application including a multi-node RAFT Distributed DNS Name Server, a Load Balancer capable of performing leader-only querying or general purpose round-robin loadbalancing, and a client application designed to evaluate performance and safety across a variety of scenarios.

Credit

For our implementation of RAFT-based DNS, we built atop an existing Open-source solution for a Distributed Key-Store implemented using RAFT called Weave. It is available at the following link. Our modifications included, fixing various errors in the leader-election conditions that the implementation used. Additionally, we added the ability to query data from the cluster using socket communication in both a blocking and non-blocking (java NIO library) fashion.

The new Classes created were:

ClientController/SocketClientController.java
ClientController/SyncSocketClientController.java
ClientController/AsyncSocketClientController.java

We also added the capacity for a node to "die". We do this by simulating the state and volatile variables that would be set when a actual failure would occur in a RAFT cluster.

Run the Experiment

To run the experiments we conducted, run the applications in the following order.

1) Run the Load Balancer

The Load Balancer Listens for Leader Election Changes. In the root/client directory, we included a start.sh script that runs the following command.

```
python client.py 127.0.0.1 5000
```

This causes the Load Balancer to listen on port 5000 for leadership changes as well as connections from the client that send metric tests through the load balancer. The Load

Balancer will perform differently, based upon the information encoded in these messages. This will be explained later in the documentation.

2) Run the RAFT Cluster

The repo contains a modification of the Open-Source RAFT-Implementation Weave. To run this version of Weave, download Apache maven using the following instructions at https://maven.apache.org/download.cgi.

To build the JAR file for the application run build.sh in the RAFT/ folder or run the following command in the RAFT/ directory:

```
mvn clean compile assembly:single
```

This will create a target/ directory with a JAR File. You will then need to modify the nodes.xml file in the RAFT/ directory. The given nodes.xml file, expects all RAFT nodes to run on localhost and will configure a RAFT node to listen on the ports that its id maps to in the nodes.xml file. The nodes.xml also contains "watchers" that are meant to be notified during leader election changes. Modify this depending on what port you run your load balancer at.

```
<WeaveConfig>
   <nodes>
      <node id="1">
         <ip>localhost</ip>
         <client>8080</client>
         <heartbeat>8081</heartbeat>
         <voting>8082</voting>
      </node>
      <node id="2">
         <ip>localhost</ip>
         <client>8090</client>
         <heartbeat>8091</heartbeat>
         <voting>8092</voting>
      </node>
      <node id="3">
         <ip>localhost</ip>
         <client>9000</client>
         <heartbeat>9001</heartbeat>
         <voting>9002</voting>
      </node>
   </nodes>
   <electionwatchers>
```

```
    <watcher>
      <ip>localhost</ip>
      <port>80</port>
    </watcher>
  </electionwatchers>
</WeaveConfig>
```

To run the RAFT cluster, open three different terminal windows, or use an orchestration software of choice.

On each terminal, pass in each RAFT node the given parameters. The first parameter is the "id" of the node. This will pull port information from nodes.xml. This second configures the cluster to listen with blocking (1) IO or non-blocking (2) IO. The third parameter is the xml configuration file to pass to each RAFT node.

This is an example of running a 3 node RAFT cluster with the above nodes.xml in blocking IO for querying mode. Each command is issued in a separate terminal window.

```
java -jar Weave.jar 1 1 nodes.xml java -jar Weave.jar 1 2 nodes.xml java -jar Weave.jar 1 3 nodes.xml
```

3) Run the Client and Test Generator

To Build a file to run tests against, run the test generator

```
python testgen.py
```

This will create a file cmds.txt. A cmds.txt is a json file, with the following format. It runs set queries and creates get queries, with a higher likelihood of calling domain names that have just been recently set. If pretty printed, it looks like the following.

```
[
 {
  "val": "94.8071926072",
  "msgid": "0",
  "cmd": "set",
  "id": "0",
  "var": "var11",
  "leader": "True"
 },
 {
  "delay": 650
 },
 {
   "var": "var11",
   "msgid": "1",
   "cmd": "get",
   "id": "0",
   "leader": "True"
 }
...
```

Then the ./start.sh script can be run to call the following command, To connect to the load balancer running on localhost at port 5000.

```
python client.py 127.0.0.1 5000
```

It will then pass the tests one-by one to load balancer who will return the performance of get / set requests and the accuracy of both as well.

To test the RAFT-based DNS, so that follower nodes are queries as well as leader nodes, convert all "leader" keys to "False, in the json stored in cmds.txt.

To do this run: `sed -i 's/TRUE/FALSE/g' cmds.txt`. The opposite FALSE and TRUE can be switched to convert a any-node querying test to a solely leader querying test.

To have the ability to kill random nodes in the cluster, create a copy of cmds.txt and name it trial[x].txt. Here x is any valid integer. Then run

```
python kill.py x
```

This will replace cmds.txt, with three kill commands placed in the test script. This will allow you to test the metrics of performance and safety in an environment where nodes can die.