Reproducing and Performance Testing Kademlia

Michaela Murray
Stanford University
murray22@stanford.edu

Isaac Westlund
Stanford University
iwestlun@stanford.edu

Guy Wuollet
Stanford University
gwuollet@stanford.edu

ABSTRACT

Distributed hash tables (DHTs) have become a widely used construct of many distributed systems, offering optimal addition/removal of nodes with minimal work and the ability to withstand significant malicious attacks. After surveying existing DHT architectures, we decided to replicate Kademlia. For our project, we both implemented Kademlia in Rust and performance tested a reference implementation on AWS. To test the ability of Kademlia to adapt to changing network conditions, we aimed to benchmark Kademlia's performance under churn under both a more realistic setting between multiple AWS instances, and a more reproducible setting using a simple single machine network simulator.

1 INTRODUCTION

Kademlia [1] is a distributed hash table that uses the XOR metric to measure distance between nodes. This results in a tree like structure that allows for quick traversal of network nodes for the purposes of retrieving stored data or storing data. As a distributed hash table, Kademlia is a key value store, that uses a key hash to locate the stored value within the network. This architecture makes Kademlia, and other distributed hash tables, resilient to changes in the network such as the frequent addition or subtraction of nodes. Our goal is to replicate the Kademlia protocol and analyze it's performance under churn.

We are submitting this project for both CS244, and CS244b. Breaking down the work, we focused on implementing Kademlia in Rust as the part of the project fulfilling the equivalent work of the 244B project and replicating performance testing figures as the part of the project fulfilling the equivalent work of the 244 project. Our group member Michaela Murray is only in CS 244B; however, as she contributed greatly to this project, her name will also be on our CS 244 paper.

1.1 Related Work

Unfortunately, the Kademlia paper only has a single performance related figure, which is based upon data from a largely unrelated study which we do not intend to replicate. Rather, the focus of the project in regards to CS244 will be to replicate the performance measurements from "Performance Evaluation of a Kademlia-Based Communication-Oriented P2P System under Churn" [3]. Churn refers to the continuous arrival and departure of participating nodes and is one of the most important factors for p2p file-sharing networks. A network that does not handle churn gracefully will struggle to be performant at a large scale and result in users flocking to other platforms.

The above study runs Kademlia over the network simulator NETHAWK-EAST. This simulator is rather expensive, and so instead we originally planned to run our tests on open-source network simulators such as Mini-net or NS-3. After further analysis of methods to evaluate the performance of Kademlia, we decided to test Kademlia on the live internet using small AWS nodes. We explored open source simulators, but found that none fit our use case exactly. Mininet is a great simulator, but is designed for the data center environment and does not well simulate a P2P network that would be deployed over the public internet in practice. Shadow is better suited to a P2P network simulation, but is very memory intensive and involves significant overhead. Instead, we decided to build a feature rich testing harness that serves as a sort of network simulator. Kademlia nodes run as individual threads and communicate without virtualizing and/or recreating the actual network. This makes the harness significantly lighter weight while sacrificing the simulation of the network itself. We thought the combination of running the both a live network on AWS and the testing harness provided the best combination of tradeoffs.

2 OVERVIEW OF KADEMLIA

As the extended Kademlia paper[1] already goes into great detail regarding the implementation and proof-of-concept of the Kademlia Protocol, we will not rehash it much in depth here. However, we will provide a brief overview of the key areas we will be focusing on in our reimplementation so as to give a reader not familiar with the original work a base off of which to understand the rest of the paper.

2.1 XOR Metric

The XOR metric was one of the defining parts of the Kademlia paper, and it defines the distance between any two nodes in the DHT to be XOR of each node's respective 160-bit ID. In the original paper, the XOR metric is shown to be a valid distance function since it satisfies the three main criterion for a distance function: 1) the distance between the node and itself is zero, 2) it is symmetric, and 3) it follows the triangle inequality. Thus, the XOR metric was proven to be a simple and cheap way of calculating the "distance" between two nodes while still satisfying all the requirements of a "real" distance function. Note that in this instance, "distance" is the distance between two nodeIDs, meaning that if similar nodeIDs are assigned to two nodes in completely different locations in the world, then they would have a small "distance" between the two nodes.

2.2 Kademlia Protocol

The basic Kademlia Protocol depends on four Remote Procedure Calls (RPCs): PING, STORE, FIND_NODE, and FIND_VALUE. The PING RPC is used often to determine whether a node is alive, which can be critical when determining whether or not to add another node to its routing table. The STORE RPC take a key and its corresponding value and then sends them into the DHT to be stored in a node with the "closest" matching nodeID to the key value where closeness is determined by the XOR metric. The FIND_NODE and FIND_VALUE RPCs both depend on a lookup process (as does STORE RPC) to find the node or the node with the desired value.

2.3 Network Structure

The network structure of Kademlia depends on each node's routing table and the underlying system of kbuckets which determines the routing table structure. Kbuckets are lists of nodes which the primary node has contacted or been contacted by at some point. There are a maximum number of 160 kbuckets that can be created, and the criterion for a particular node falling in one kbucket or another is the XOR metric distance between that node's and the primary node's IDs. This network structure allows for flexible growth of the routing table throughout the course of the DHT and also for concise compartmentalization of different nodes.

3 REASONING BEHIND USING RUST

One of the defining characteristics of this project is the choice to use Rust as the prevailing language for the implementation of our Kademlia Protocol. Rust has many advantages, but the three main reasons why we chose Rust for this project is its 1) ownership model, 2) high performance comparable to C, and 3) ease of use. We will go into each of this reasons briefly so as to highlight why we made this critical choice in our project.

3.1 Ownership Model

One of the outstanding features of Rust is its ownership model. This enables programmers to eliminate an entire class of security bugs which occur due to memory mismanagement. Thus, it is an especially attractive language to write or rewrite large, complex systems in.

3.2 High Performance

Rust is a high performance language that is comparable to C in many cases. This makes Rust even more desirable combined with the ownership model mentioned above.

3.3 Ease of Use

Rust was created for much greater usability as compared to other low-level languages like C. Rust emits easy readable and easy to reference compiler errors which allows the user to have an immediate direction to follow when they run into a compilation error. Between the easy-to-read compiler errors and extensive documentation, Rust is better than languages such as C when it comes to addressing errors in the program.

4 IMPLEMENTATION

Below, we discuss the implementation of the Kademlia protocol and the test harness below.

4.1 Kademlia Protocol

The Kademlia Protocol has four main parts: Basic Structs, Overlay Network, Protocol RPCs, and bootstrapping into the network. The basic structs consists of Nodes, ZipNodes, and RPCs. For simplicity's sake, we represent node IDs as Rust unsigned 64-bit values rather then the larger unsigned 160-bit values used in the paper. In our implementation, node IDs are initialized from a hash on the node IP. Node IP's are initialized to a string passed at node creation. The Nodes struct describes each node within the Kademlia Distributed Hash Table and is comprised of the k-buckets table, a node ID, an IP, a port, a lookup_map, a lookup_counter, and a storage HashMap. Our k-buckets table is represented as a vector of vectors where the i^{th} inner vector corresponds to the "i^{th} bucket" containing all nodes whose xor distance is within $[2^i, 2^{i+1})$

ZipNodes are a compressed version of a Node struct, only containing an ID, IP, and Port number. Each holds its k-buckets in a Vector The Protocol RPCs are the same four described above in Section 2.2. [INCLUDE STUFF ABOUT BOOTSTRAPPING]

4.2 Test Harness

The testing harness is meant to simulate a basic network such that we can test the Kademlia RPCs without any extra complexities such as DNS lookup and NAT traversal. In the testing harness, each Kademlia "node" is represented by a thread which will come from a queue of Kademlia RPCS. We have created additional global data structures that serve as the shared "network" for threads to communicate.

4.3 Conversation with Author

As Professor Maziéres was one of the authors on the Kademlia paper, we were lucky enough to get some of his advice on our project. He suggested the one of the hardest problems for implementing Kademlia, and peer to peer systems over the modern internet, is NAT traversal. In order to fix this problem he suggested that we could simply assume all nodes in the network used IPv6 addressing. Thankfully, with AWS we are able to bind a static IPv4 address to each node we create. This will allow us to either make the IPv6 assumption or to make the assumption that nodes in the network are not behind an IPv4 NAT. The testing harness also avoids the NAT traversal problem as it does not virtualize or simulate the network itself.

5 PERFORMANCE TESTING

5.1 AWS

5.1.1 Performance Metric. The metric for our performance testing was success rate. Success rate is defined as the portion of get requests that return the same value as originally set for a given key. More specifically, we tested success rate for various level of churn. Churn is how often nodes leave the network. The level of churn is measured by the mean uptime of each node. A shorter mean uptime means more churn.

5.1.2 Setup. We attempted a loose reproduction of [3]. There were some significant differences. [3] used the NETHAWK East simulator whereas we ran a live network on AWS. The paper also used 400 nodes whereas our implementation was only able to run 20 due to AWS limits. Further, [3] used a different underlying messaging protocol from the original Kademlia paper, and we used a reference implementation [2].

Inspired by [3] we made several assumptions. First we used a single Kademlia server that never churned to bootstrap new nodes or nodes reconnecting to the network after churn (failure). We modeled the uptime for each node as an exponentially distributed random variable and set the mean uptime and downtime equal for each node. This resulted in a system where half of all nodes are live in the network at any given time in expectation.

In order to performance test Kademlia, we chose to run many AWS EC2 instances. Each instance used a shared Amazon Machine Instance (AMI) with the Kademlia code, and 4 scripts. One script created an initial node in the Kademlia network, a second script bootstrapped a Kademlia server given a list of existing Kademlia servers, a third script bootstrapped a Kademlia server and made a set request for a given key value pair, and a fourth script bootstrapped a Kademlia server and made a get request for a given key.

Each instance had a bound public IPv4 address that was used to connect with other instances. Kademlia ran over port 8468. In order to coordinate the instances and run performance tests, we used a central controller script running on a personal computer. This script used the AWS Python SDK, boto3, and the AWS command line interface. It created instances and then used AWS Systems Manager (SSM) to send commands to each instance. SSM runs an endpoint agent on each instance that servers as a proxy user accounts and run commands on behalf of the script. Alternatively, an SSH client could have been used to issue commands. Due to poor internet connectivity the pipe often broke when attempting to use SSH. Since SSM is asynchronous, we did not need to worry about maintaining a constant connection for long running performance tests. Due to AWS limits, we were only able to run at most 20 Kademlia nodes at a time. We attempted to raise the limit, but were unsuccessful.

5.1.3 Results. We attempted to reproduce Figures 1a and 2 in [3]. Despite the differences between our setup and [3], we achieve relatively similar results.

For 1 we see that Kademlia provides a higher success ratio for lower churn. Note that the x axis displays the the mean uptime and mean downtime of nodes in the test. That uptime is an exponentially distributed random variable calculated at the launch of each node. For a lower mean uptime, the node will churn more often, leading to a lower success rate.

A lower mean uptime, and thus higher churn, leads to lower success rates because this means entries in the routing tables of each node and out of date and even that the stored value itself might no longer be in the network. Our graph shows this result as expected.

For reproducing Figure 2 in [3] we varied the K value in Kademlia and measured the resulting success rates. With Alpha = 1, we would expect higher K to lead to higher success rates at all levels of churn. This is because higher K means larger routing tables for each Kademlia node and a greater likelihood that some entries in that routing are still fresh, meaning the associated nodes are still running. This result is demonstrated in 2. However, increasing K above 2 seems to have a de minimus impact on the success ratio for a given level of churn. This is likely because at least one entry in the routing table is still fresh.

5.2 Test Harness

Performance

6 CHALLENGES

No systems project would be complete without a series of challenges facing it. Below, we briefly overview some of the challenges we faced in our project and how we overcame them.

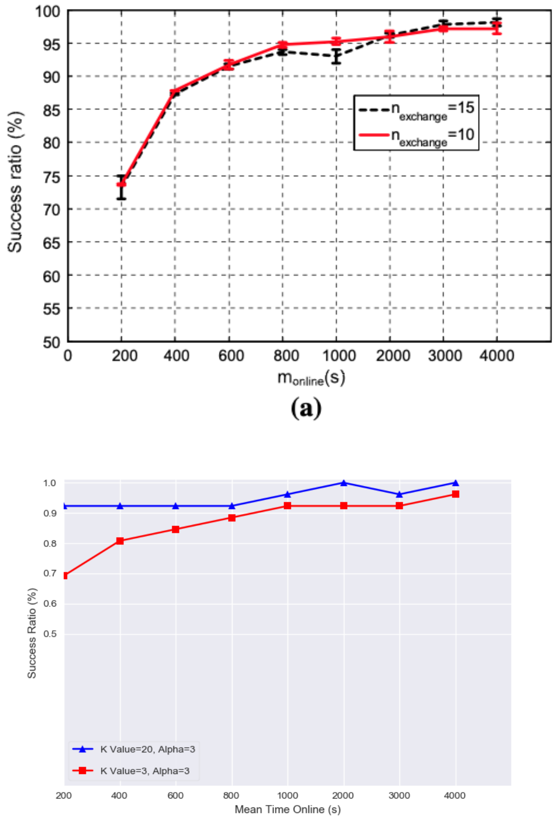

Figure 1: The top graph is Figure 1a from [3]. The bottom graph is our loose reproduction. The top graph has K Value = 3 and Alpha = 3. The original Kademlia paper suggests a K Value = 20 and Alpha = 3. The $n_{exchange}$ is the top graph is a reference to a different messaging protocol used in [3] and not relevant to the reference implementation of Kademlia. The x axis is the mean uptime and mean downtime of each Kademlia node in the network. This means that as the x axis value increases, churn decreases. The y axis is success ratio, which is the performance metric. We expected increasing success ratio for decreasing churn, or an up and to the right graph.

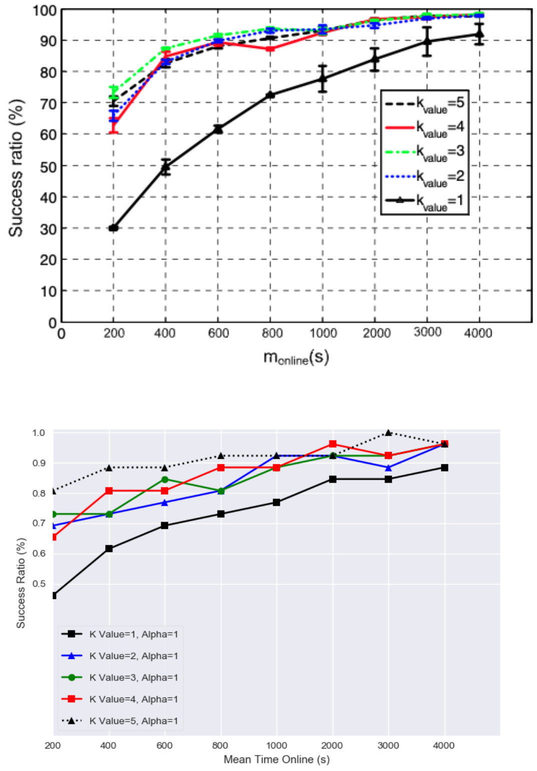

Figure 2: The top graph is Figure 2 from [3]. The bottom graph is our loose reproduction. The top graph has Alpha = 1. The x axis is the mean uptime and mean downtime of each Kademlia node in the network. This means that as the x axis value increases, churn decreases. The y axis is success ratio, which is the performance metric. We expected increasing success ratio for decreasing churn, or an up and to the right graph. For larger K Values, we expected an increasing success rate as K is a system wide replication hyper parameter in Kademlia.

6.1 Rust

While Rust has many benefits as detailed above, it also has quite a few challenges. It is generally agreed upon that Rust is a hard language to get used to, and while its ownership system is one of the best parts of Rust, this also makes it harder and less satisfying when starting out. Also, since Rust is a relatively new language, there may be some features which are still in the prototyping phase and do not have the extensive documentation other features may have. However, through perseverance, reading, and trial-and-error, we were able to adjust to the Rust ecosystem relatively well.

6.2 Protocol and Test Harness

The test harness and protocol implementations presented many challenges, some related to the challenges with using Rust, as outlined in the previous section.. One primary section we had difficulty defining was the lookup algorithm. The lookup functionality could not be contained in one simple recursive function but instead had to be split up into a three-part state machine to accommodate the RPC structure. Thus, debugging and ensuring that the correct kbuckets were being queried/accessed presented challenges throughout the development process. As for the test harness, some difficulties included determining what setup would lead to the easiest to debug interface while still trying to simulate a realistic system.

6.3 AWS Setup

The AWS Setup presented many challenges. First, using a custom protocol requires a specialized security group on each AWS instance. The default security group on AWS is listed as accepting all connections, from any protocol, over any port. However, for security reasons the security group rejects all connections except for SSH. This was quite challenging to discover. A simple solution was to redefine the security under a different name to accept all incoming network connections over all ports.

Running commands on multiple instances was also troublesome. The best existing solution seems to be AWS Systems Manager (SSM). Using an SSH client is an option, but was quite challenging in practice for long lived connections on many machines. This could also be the result of a spotty internet connection and the increased internet traffic during Covid quarantine. SSM is asynchronous and helped solve this problem. However, SSM has its own problems. SSM runs an endpoint agent and requires a custom defined Identity and Access Management (IAM) policy for any EC2 instances that accepts SSM commands. This is also not well documented on AWS and had to be learned through trial and error.

Regions in AWS were also a challenge. It was very difficult to get a single script using the AWS SDK to interact with EC2 instances running in multiple regions. This seemed to be the result of a credentials issue. The SDK state is region specific. The desired region is generally specified in a config file stored in the root directory of the computer. The AWS CLI helps set this up with a simple config command. However, the region configuration, and account ID and secret key, can be set as environment variables of the shell. When changing region in a script using the SDK, the config file on disk does not change. After many failed attempts to work with instances in multiple regions, I suspect the environment variable of the shell conflicts with the config file on disk. For this project, we ended up using a single region in AWS.

FUTURE WORK

There are many additions and enhancements we can make to our current implementation. The first set of enhancements is optimizing the lookup algorithm. The Kademlia paper details various optimizations like accelerated lookup. We also originally aimed to add Byzantine Fault Tolerance to the Kademlia DHT.

REFERENCES

[1] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.

[2] Brian Muller. 2020. bmueller/kademlia. (2020). https://github.com/bmuller/kademlia.

[3] Zhonghong Ou, Erkki Harjula, Otso Kassinen, and Mika Ylianttila. 2010. Performance evaluation of a Kademlia-based communication-oriented P2P system under churn. *Computer Networks* 54, 5 (2010), 689 – 705. https://doi.org/10.1016/j.comnet.2009.09.022