Willow Cutting Distributed Hash Table

Joon Sung Park, Jacob Carstenson
Computer Science Department, Swarthmore College, Swarthmore, PA 19081

May 12, 2016

Abstract

We present Willow-Cutting Distributed Hash Table, a novel implementation of a distributed hash table that is designed to efficiently locate key value pairs in the distributed system. The hash table algorithm that serves as the backbone of this implementation, which takes a similar approach as the extensible hash table algorithm, grants the hash table a set of unique properties that can be exploited by a distributed system. These properties include no rehashing of keys during resizing and a key range of the hash table. Because any given key will not get rehashed (that is, changed) in its lifetime, Willow-Cutting Distributed Hash Table can keep track of the node that a given key value pair is in simply by maintaining a routing table that records a nodes IP address and the hashed key range that node is responsible of. This gives a significant advantage for maintaining Willow-Cutting Distributed Hash Table efficient and elegant in its implementation. Theoretical analysis of the system, as well as a number of rudimentary simulations, suggests that Willow-Cutting Distributed Hash Table is likely scalable, and elegant in its way of implementation.

1 Introduction

A distributed hash table, as its name suggests, distributes key and value pairs over multiple hash tables that are often in different nodes. Despite the extra cost it takes to transfer key value pairs from one node to the another, distributed hash table has remained one of the most studied, and implemented ways of resource sharing in data intensive settings due to its potential for efficient distribution of dictionary-like data. However, there are challenges to building a truly powerful distributed hash table; one of the greatest challenges in designing a distributed hash table is organizing the hash table over multiple nodes so that any node can efficiently keep track of which key is contained in which node.

In this paper, we answer to this challenge through a novel implementation of distributed hash table that utilizes Willow-Cutting Hash Table – an original hash table designed by one the authors. Willow-Cutting Hash Table was originally designed to avoid rehashing during the resizing process of a hash table, and by avoiding expensive rehashing, to optimize the hash table's performance. But this property of one-time-hashing also came with an extra benefit of guaranteeing each key value pairs an unchanging hashed-key during the pair's lifetime. Our distributed hash table, Willow-Cutting Distributed Hash Table, utilizes this property of unchanging hashed-key to organize hash table over multiple nodes.

In Willow-Cutting Distributed Hash Table, each node is represented as a Willow-Cutting Hash Table. When a key value pair is inserted into Willow-Cutting Distributed Hash Table, its key is

hashed. Then the distributed hash table places the key value pair to appropriate node based on the hashed key. To maintain the location of the key value pair, the distributed hash table simply maintains a routing table dictating which node is responsible for a given hashed key. Simply maintaining a routing table is sufficient for the distributed hash table to keep track of where a given key is since the key's hashed key is permanent during its lifetime.

The overall results of both Willow-Cutting Hash Table and Distributed Hash Table are promising, with pointers to numerous potential future works for improvement. In the next section, we give a quick overview of related works done on the topic of distributed hash table. In section 3, we will walk through the more detailed protocol for Willow-Cutting Hash Table and Distributed Hash Table, and provide some important implementation details. In section 4, we provide numeric performance results for our implementation, and in section 5, our interpretation and conclusion of our works done. In the last section, we provide some possible future directions and extensions that have come through the course of our research.

2 Related Work

Willow-Cutting Hash Table is a relatively novel approach, but it is not the first attempt to make a more dynamic hash table. For example, linear hash table attempts to minimize the downtime caused by resizing by gradually increasing the bucket capacity. On the other hand, extendible hash table utilizes a trie for bucket lookup on bit string hashes to allow faster and gradual resizing [1]. Our implementation will work to give a similar kind of resizing that is not necessarily as gradual, but is similar in that resizing is in place and does not require extra memory to complete. Nonetheless, Willow-Cutting Hash Table offers a unique design and perspective to previously popular suggestions for dynamic hashing.

The topic of distributed hash table is also a widely studied one. In particular, there have been numerous attempts to create a distributed hash table that is scalable over many nodes. To address problems of scalability, a second generation of P2P applications appeared including: Tapestry, Chord, Pastry, and CAN. After a huge spike of interest in peer-to-peer distributed hash table in the aftermath of Napster, these four algorithms are some of the early results of the field of distributed hash table. These algorithms are still very influential today, and became important guide for us in understanding the popular makings of distributed hash tables. Tapestry in particular is interesting because it maintains a routing table data structure of each node's closest neighbors to route requests closer to where the data resides. When a new node wants to join the DHT, the new node obtains a randomized node ID and figures out its closest neighbors to this ID and broadcasts to them its existence. Then, the other nodes will send the new node a temporary routing table for which the new node will use to conduct an iterative nearest neighbor search to fill in its final routing table [2]. Our implementation of a new distributed routing table is similar in that a routing table is maintained for each node, but this time it is representative of the entire set of nodes. A broadcasting model of propagating the global routing table is also used to spread this information as well.

3 Willow Cutting Distributed Hash Table Implementation

All of the code written was done in Python to make the tight schedule we had a little easier.

3.1 Hash Table

As mentioned previously, the main goal for designing Willow-Cutting Hash Table (WCHT) was to avoid rehashing of the elements during resizing process. This would theoretically make the resizing process much more efficient, and make the overall performance of hash table much more predictable. WCHT successfully accomplishes this by chaining to a special type of binary search tree called Partition-Driven Binary Search Tree to represent its buckets.

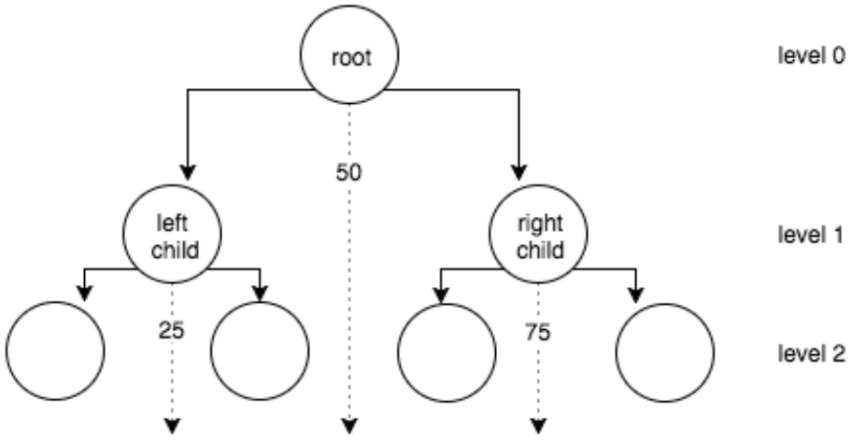


Figure 1: PDBST with Range 0 to 99

3.1.1 Partition-Driven Binary Search Tree

Partition-Driven Binary Search Tree (PDBST) is much like a regular binary search tree in that each element in the tree has two children, and maintains a certain degree of order. However, PDBST is different from a regular binary search tree for two main aspects. First, it has a set range (for example, PDBST shown in Figure 1 has a range to 0 to 99), and cannot accept any keys that is not within its range. Second, when determining whether a given child should go to the left or right sub-tree, PDBST uses the values that "partition" the capacity of the tree into half rather than comparing the new key to the parent's key.

In the case of Figure 1, for example, the first element that is inserted to the tree will be the root of the tree. When adding the second element, however, PDBST will check if the new element's key is greater or smaller than the value 50, which partitions the current PDBST's range in half. When the given new key is greater than 50, it will be placed in the right child of the root, and if smaller than 50, it will be placed in the left child of the root. We have designed the upper and lower bounds of the PDBST range to be inclusive, and this is true of all the other data structures we have implemented.

Note that PDBST will take the same logarithmic time for its basic operations (PUT, GET, DELETE) as the concept of halving the number of elements at each level of subtree remains

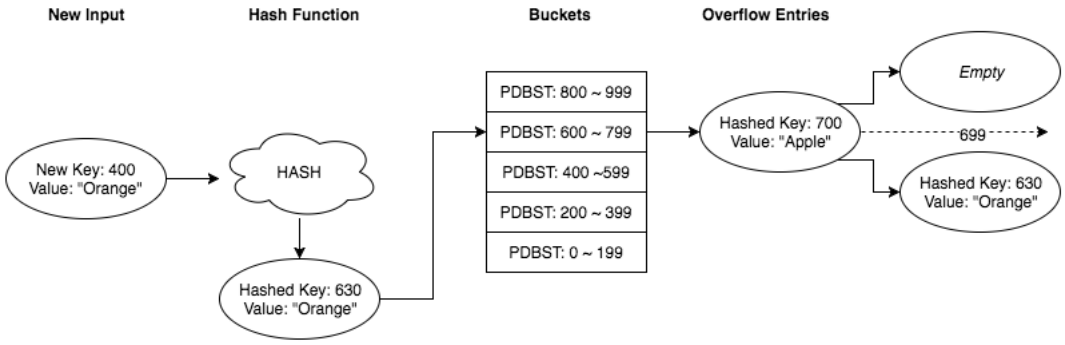


Figure 2: An Example of WCHT PUT Operation

unchanged.

3.1.2 Willow Cutting Hash Table

At its core, Willow-Cutting Hash Table (WCHT) is a chained hash table with PDBST working as its chained data structure. It is similar in overall design to a more conventional chained hash table, but it takes a full advantage of the PDBST's properties to more efficiently resize its buckets.

As a chained hash table, WCHT's buckets are each represented as a PDBST. And much like PDBST, WCHT also comes with a set range. The range of each PDBST in the hash table is determined by the number of buckets present in the hash table and the overall range of current WCHT. For example, in Figure 2 is a WCHT with bucket capacity of 5, with overall range of 0 to 999. The range of each bucket (PDBST) is in turn determined as 0 to 199, 200 to 399, and so on. When an input key is hashed, unlike in a regular hash table, the key is hashed to a unique integer that is within the overall range of the WCHT, and then put into the PDBST bucket whose range contains the hashed key. Conceptually, this is essentially imitating the process of perfect hashing (no-collision hashing). This is because in the case of WCHT in Figure 2, although there are only 5 buckets present, each integer within the range of 0 to 999 can be seen as a bucket without actual physical memory allocation.

To resize this hash table, that is, to increase the number of physically allocated buckets, WCHT has to get the left and right subtrees of the root of each PDBST, and make each subtree a new bucket. After that, all WCHT has to do is insert back the root node. This way, WCHT almost entirely avoids rehashing of its elements during the resizing process, perhaps with the exception of the root in each PDBST.

3.2 Distributed Hash Table

Here is the main problem of the project: how does one effectively distribute a Willow-Cutting Hash Table (WCHT)? The most important aspect of this problem is locating the right node who contains the data you would like to operate on.

3.2.1 The Protocol

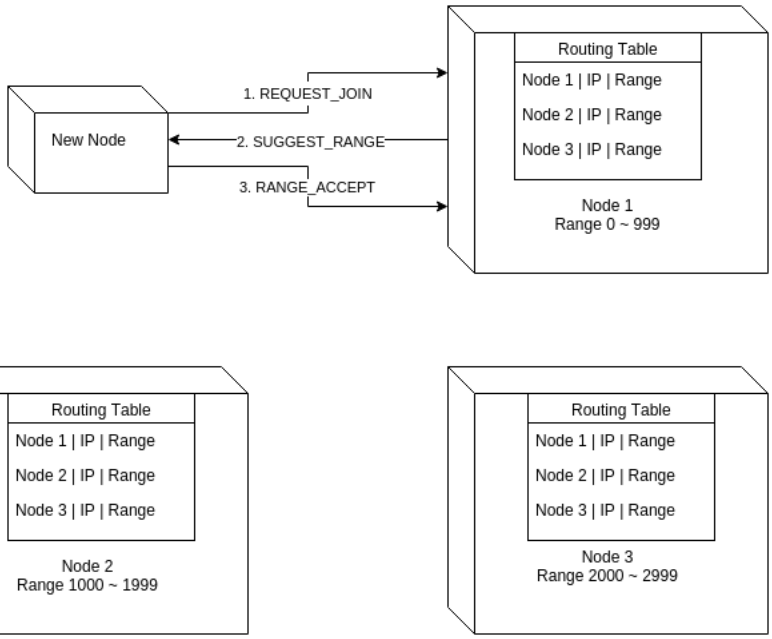
In order to understand how the Willow-Cutting Hash Table was distributed, one must first image a global hash table, which has a given global range. This table can be split up into subsections (containing their own ranges) which are dispersed among the nodes participating in the Distributed Hash Table (DHT). Theoretically if a client wants to access a value in this global hash table, it needs to contact the node which has the range of the desired value.

Moving away from the theory of the DHT, there are two sets of programs made to realize this. There is a server program which acts as the node itself, which has peer-to-peer qualities but has no way to interact with the data it is taking care of. This is where another client program is needed which takes care of interacting with the hash table by contacting these servers to request an operation. The server consists of a main server factory (explained in section 3.2.2) which maintains the WCHT and a routing table representation of the global hash table. Nodes can only be added sequentially, meaning the first node to participate creates a WCHT which *is* the global hash table. When other nodes want to join, they must connect to a participating node and request to split the WCHT of that specific node. With this protocol, nodes may only be added one at a time, and should contact the node with the highest range in order to maintain an even distribution. When a new node wishes to join, the global routing table must be updated to reflect the shortened range of the split node, but also the IP address and range of the new participating node. In order for all other nodes to know about this change, a broadcasting model is implemented where the split node also sends this new routing table to all of the other nodes. Since each key needs to be hashed before deciding where among the nodes it should be found, each node needs to know the global upper and lower bounds of the global hash table because these values are needed to give a consistent hash value to

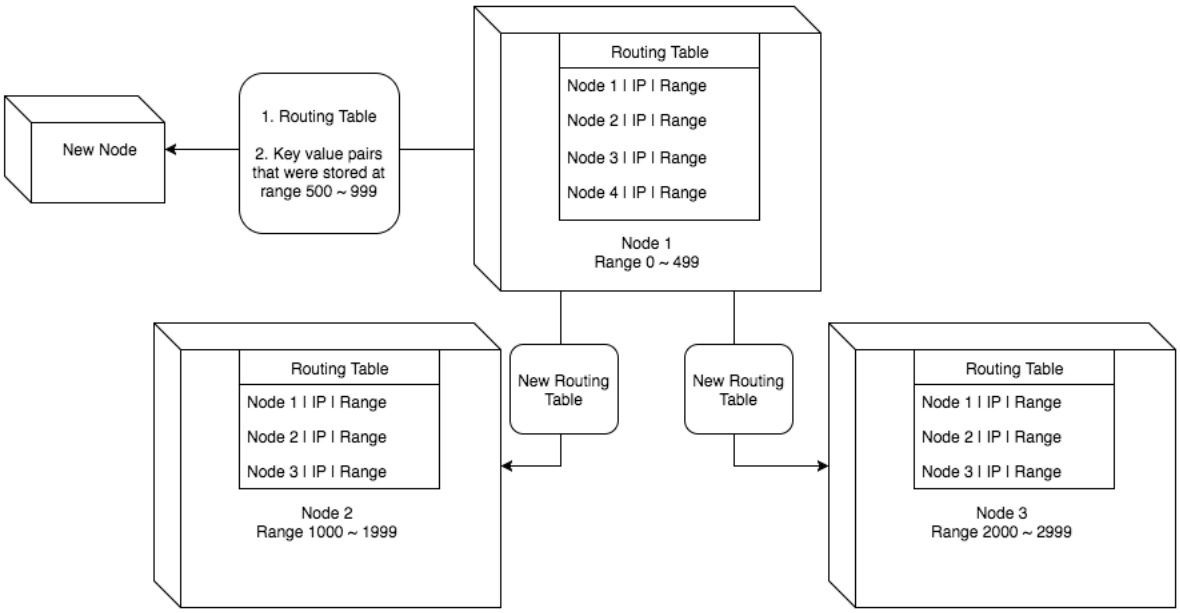
The client implementation is much simpler and more sequential than the server. Here one client factory is created which connects to a given node's IP address and requests a certain operation (get, put, or delete) to be done on the global DHT. Since the client has no knowledge of the underlying routing table, some requests for keys that lie in other nodes will require the request to be resent to a different node. The first node will send back the IP address of this correct node. Finally, the client will print the results of the operation.

3.2.2 Twisted

To distribute the Willow-Cutting Hash Table, a design choice was made to use socket based network programming as a form of message passing between nodes, which makes deploying the servers and clients across machines easier. Twisted is an event-driven network programming module written in Python which was used for this implementation. At the core of Twisted is a factory which either listens for connections on a port if it is a server or creates connections to a given host if it is a client. Once a connection is made, each individual connection will spawn a protocol object which handles the main protocol of messages sent. Not only does this abstract away the tedious aspects of creating and binding sockets, but it also separates persistent data that is maintained



(a) A new node requests to join the DHT



(b) This added node obtains a chunk of Node 1's hash table and the routing table is broadcasted

Figure 3: The protocol for when a new node joins the DHT

across connections, which the factory takes care of, from data that is relevant only to specific connections, which the protocols cover. Multiple of these factories can be running at once, and this is how each server node will act as a peer, by listening on a port and initiating connections to other nodes when the protocol requires this.

4 Results

4.1 Willow Cutting Hash Table

Overall with our current implementation, a chained hash table with binary search tree has an upper edge in general operation performance against WCHT. However, in terms of resizing operation alone, WCHT is definitively faster than a chained hash table. It is also noteworthy that the optimal loadFactor (size/capacity) for WCHT is much higher than a regular hash table. These points will be further expounded and the possible explanations for the result will be discussed in the next subsections.

4.1.1 Optimal Performance LoadFactor

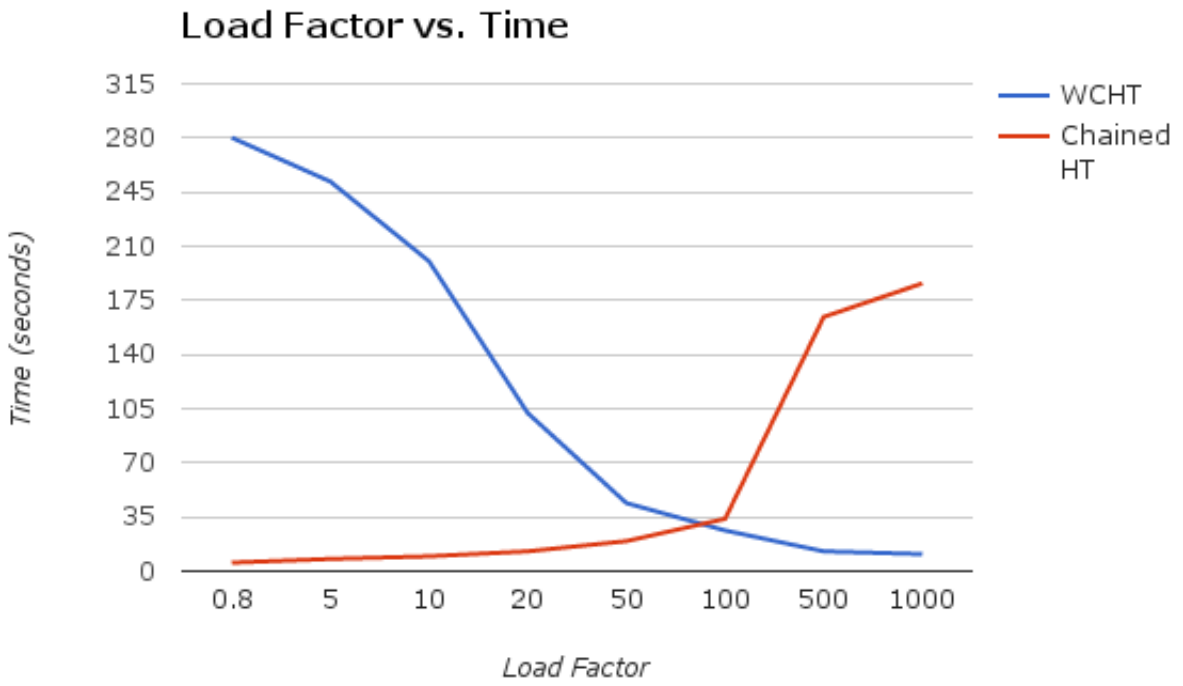


Figure 4: The plot of loadFactor vs. time for WCHT and chained HT on 250,000 PUTS

LoadFactor of a hash table is size/capacity, where size is the number of elements in the table, and capacity is the number of buckets present. Therefore, the loadFactor effectively determines

Chained HT (0.8 LoadFactor)	WCHT (1,000 LoadFactor)
5.80 Seconds	11.65 Seconds

Table 1: Operation Time for 250,000 PUTS

when and how often a hash table will resize; simply put, the higher the loadFactor, the less frequent resizing occurs.

Figure 4 shows the performance for 250,000 PUT operations on WCHT and a regular chained hash table on different loadFactor. Whereas a chained hash table functions at its best when the loadFactor is low, WCHT functions the best when the load factor is relatively high. Table 1 above, records the optimal performance loadFactor and their operation time for 250,000. The optimal loadFactor for the chained hash tale was 0.8 whereas the optimal loadFactor for WCHT was 1000. This is quite a large difference and suggests that WCHT performs at its best when it is of high density compare to a chained hash table. This is an expected result that we accredit to the fact that WCHT benefits the most for its performance in its ability to split its buckets and not rehash. When there are many elements in each of the buckets (when the loadFactor is high), there are more elements it can split into two chunks, whereas when there are very few elements in each of the buckets (when the loadFactor is low), there are less elements it can split, and therefore it benefits less from its efficient resizing.

Another notable result in Table 1 is that with our current implementation, chained hash table performs better than WCHT at their respective optimal loadFactor. However, the overall result is nonetheless promising for a few reasons. On the average, the chained hash table took about 15% of its total operation time for resizing (for example, when loadFactor was 10, it took 2.0 seconds for resizing and 9.6 seconds for total operation for 250,000 PUTS). However, WCHT on the average took only about 1% of its total operation time for resizing (when loadFactor was 10, it took 1.7 seconds for resizing and 200 seconds for total operation for 250,000 PUTS). That is, the overall proportion of time spent for resizing is much less for WCHT than it is for a regular chained hash table. In fact, as it will be looked into in more detail in the next section, the resizing process is definitively faster for WCHT compare to a regular chained hash table given the same loadFactor and operation. This suggests that WCHT's resizing algorithm is definitely much more efficient than a regular resizing of a hash table that involves rehashing of the elements.

Our current code for WCHT is far from being optimal and we believe that with more refined understanding of the Python language and further optimization, the general performance of WCHT can be largely improved. With the resizing process as efficient as it is, further optimization may bring more impressive performance from WCHT.

4.1.2 Resizing Runtime Isolated

Load Factor	Chained HT	Willow-Cutting HT
10	2.00	1.757
1000	19.93	0.0113

Table 2: Time taken for resizing for 250,000 PUTS

Table 2 records the isolated time for resizing process that has taken in chained hash table and WCHT with two different loadFactor. Even for the lower loadFactor, it is evident that WCHT is faster for resizing its buckets than regular chained hash table. The improvement, however, is much more exaggerated when the loadFactor is greater for the reasons stated in the previous section.

4.2 Willow-Cutting Distributed Hash Table

Nodes	Average Time (s)
1	1.64
2	1.82
3	1.87
4	1.83

Table 3: 1000 Puts into various sized WCDHTs

Using WCHT as the basis, we managed to implement a working distributed hash table. Due to the relatively unique nature of our distributed hash table, however, it is difficult at the moment to quantitatively conclude the runtime quality of its performance. We also acknowledge here that our broadcasting model of Willow-Cutting Distributed Hash Table is likely not the most scalable model and implementing a more efficient way to propagate the routing table is necessarily going to be one of the first improvements that need to be made for the distributed hash table to become more scalable. Thus what we present in this section is largely a prototype that could be worked towards building a more robust distributed system with greater potential.

Figure 3 records the time for 1000 PUT operations into Willow-Cutting Distributed Hash Table with different number of nodes in the system. One quick thing to note on the small scale testing (1000 PUT instead of a much bigger number of PUT) is that Twisted proved to be a large limiting factor to the DHT despite how easy and pretty it made the code. The main issue with Twisted is that it had a limit to how big you can send a string from one machine to another. This effectively placed a size limit on how big our hash table could be because splitting would cause an error. We found that limit to be approximately 5000 values if the hash table was filled before splitting.

These limitations gave rise to a test to be conducted on the Willow-Cutting DHT: The performance of putting values over varying sizes of connected nodes. Since our global hash table had to have a max range of 0 - 5000, this allowed for a maximum of 4 connected nodes since each node needed to have at least 4 underlying WCHTs in order to split. And with the maximum put limit

to be 1000, the test involved putting the values 0-999 into the WCDHT with nodes varying from 1 to 4.

The table 3 shows the results of this test. To obtain the average time, 5 of these 1000 puts were run for each size of the DHT, where the client sends these requests to its localhost server. The data shows there is a slight decrease in performance from one node to two, but not much difference for the larger sets. This is most likely because for one node, no forwarding is required, but for more nodes only one there is a maximum of one forward that the client will need to perform to get to where the key is located, because the broadcasting model ensures all nodes have up to date routing tables. While further work is needed to provide a more robust result, the result we provide in this section showcases to a certain degree the efficiency of maintaining a routing table. Because routing table provides a relatively direct path to the node that contains a certain key, the average runtime does not increase by far even when the number of nodes in the system increases.

5 Conclusions and Future Directions

In this paper, we presented a novel design and implementation result for hash table, as well as preliminary implementation result for distributed hash table that is based on WCHT. The overall result is promising with WCHT showing clear advantage over the run time performance for resizing, and with the distributed hash table suggesting signs of efficiency for maintaining a routing table with hashed keys. However, there are clear limitations to our current implementations that were discussed in this paper and as our conclusion, we provide some insights into possible future improvements on Willow-Cutting Distributed Hash Table. But nonetheless, we hope our result so far serves as a good starting point, a demo and a prototype, for a possibly much more robust and meaningful distributed data structure.

First, it has dawned on us that Python and Twisted Web Framework may not be the best tools for building our implementation of hash table and distributed hash table. The reason is two folds. One, it is unclear what kind of in-house optimization takes place within Python language, and for building an experimental data structure, Python seems to abstract away too many important details a developer would need for building clear and correct data structure implementation. Two, Twisted Web Framework is an event-driven framework that centers around single infinite loop that waits for an incoming action. Willow-Cutting Distributed Hash Table, due to its numerous communications with multiple nodes, is better suited for a back-end that would provide multiple waiting loops as well as the power to stop and restart the loop, none of which is provided by Twisted. Therefore, we believe the immediate next step to this project is to build a more robust version using C and its socket frameworks, which would provide a higher degree of control needed for the distributed hash table that Python and Twisted does not provide.

Second, the current broadcasting model of distributed hash table can be improved so that it is much more scalable over greater number of nodes. One of the most obvious ways to achieve this is to make the propagation of routing table lazier. Broadcasting information over the entirety of the distributed system is extremely costly, and makes the system not scalable. Lazy model, on the other hand, is meant to send the needed information when it is needed. In the case of Willow-Cutting Distributed Hash Table, this would mean sending the updated routing table to another node only when that other node needs to know the updated information of the routing table. There are possibly many ways to achieve this, but it is likely necessary that the broadcasting model replaced with a more scalable one.

Third, in this paper, we have not dealt much with node failure and fault tolerance. This is largely because of the scope and resources available for this research, not because of its lack of importance; any robust distributed system should be able to answer efficiently to the challenges posed by node failures and other faults that could occur. The protocol that deals with failure has to be developed for the current Willow-Cutting Distributed Hash Table.

6 Meta-Discussion

Throughout working on this project, there were aspects that were easy, and some that were hard. The most difficult part of this project included implementing the Willow-Cutting hash table which included a lot of unforeseen overheads, like hashing, or how to deal with the range that is so inherent to the PDBST. It was a challenging task to write our code such that it is optimized as well as it could, especially with the WCHT and its distribution, we have a lot of code that is not the most elegant or straightforward, like how we decide to split the hash table or places where list comprehension could have been used. Twisted was also a huge road block in implementing the distributed hash table not only because of how hard it was to grasp asynchronous programming, but because of the little quirks and errors that Twisted created that we had to work around. Despite its downfalls, Twisted did make our code easier to write by avoiding socket programming and non-blocking I/O, and making the main protocol very clear. Other aspects of the project that were easy were using inheritance to build up all of our data structures starting with the base DataStructure class, as well as the rudimentary data structures such as BST, linked list, and PDBST.

On the whole, our project did not stray very far from what our original plan was: to implement a hash table based on the PDBST data structure and effectively distribute it. Some smaller side plans did not get implemented because of time constraints which include not implementing a lazy model opposing the costly broadcasting model and creating a chained version of the DHT to have better comparisons. Overall this project proved to be very difficult yet rewarding to see the results.

References

[1] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, September 1979.

[2] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J.Sel. A. Commun.*, 22(1):41–53, September 2006.