An Implementation of the Chord DHT

Daniel Okazaki
dtokazaki@scu.edu
W1053784

Stephen Pacwa
spacwa@scu.edu
W1113743

Goals

A distributed hash table (DHT) is a structure that allows for a file or reference to be found that is stored in a node inside a peer-to-peer network. A distributed algorithm known as a routing overlay facilitates the communication of the messages needed to find and retrieve the file or reference in the system [1]. They generally work by building an overlay structure that allows any node to hop along many different peers until the node that contains the item in question is reached. There are generally two approaches to the design of these networks, they are either structured or unstructured. In a structured network, there are specific requirements surrounding how requests are made and how the topology has been formed whereas in an unstructured network, nodes can join anywhere in an ad-hoc manner [1]. For the purposes of our project we will be focusing on a structured network.

Our goal is to develop the Chord DHT in order to operate as a lookup for document metadata, but it should be able to operate as a general system that is able to store any metadata with a key. We will also build a small program that will seed the network in order to allow us to make several queries and test the validity of our program and its effectiveness.

Challenges Addressed

A typical server/client architecture is the most basic implementation of a file to location mapping table. In this architecture, a server serves as the single source of all key to location mappings. The server is responsible for managing the locations of each file, which involves first storing the key to location values, as well as making sure that these mappings are held up to date at all times in case of changes in the network. All requests to add a file, remove a file, or search for a file are all sent to this server. The biggest flaw with this type of system is scalability. The centralized server must scale linearly with the amount of requests that it gets, meaning that new hardware must be added to handle incoming requests when the bandwidth of the server reaches maximum capacity. Upgrading vertically can only go so far, as adding better and better hardware to a system will eventually reach a point where performance is capped. Upgrading horizontally also introduces new problems with consistency and concurrency. Adding new servers to help handle the load means that locking mechanisms need to be put in place so that transactions modifying

the mapping table are done serially in order to maintain consistency. This also creates a single point of failure on the server that stores the entire mapping table. In order to prevent this, the mapping data can be replicated, however this also increases the complexity of the overall system. If one of the main servers goes down, the amount of bandwidth and performance of the overall system is drastically reduced and in the worst case, completely incapacitates the system through the single point of failure.

A distributed hash table attempts to solve these problems by distributing the workload of the mapping information to all its nodes. This prevents any one node from disabling the system or leaving it in an inconsistent state by going offline. This also distributes the workload of handling requests throughout the network, making sure that the only nodes who need to be contacted, are the ones that help the original node find the mapping information. Mapping information is kept up to date as files are initially mapped to exactly one node, and is also replicated to a small subset of the network called successors of that node. Node joins and removals are handled by a stabilization algorithm that runs periodically, and only handles the nodes in the successor list of a particular node. A centralized solution would have to either constantly send pings to all of the storage nodes to tell if they were online, or reach an error on a request. In the later case, the server wouldn't necessarily know if the request failed due to network error, or due to the storage node being down, meaning that a failure detection algorithm needs to be run for every failure encountered to find the cause and repair the failure.

Benefits of Selected Approach

A distributed hash table is able to dynamically find files and update itself in O(log *N*), where *N* is the number of total nodes in the system. This speed is achieved through each node holding a finger table of B nodes, where $2^B = N$. The finger table holds node identifiers in ascending order, where the highest node in the system wraps around to the lowest node's identifier in the system forming a ring.

Files searches occur as follows. The filename is hashed, and the first *B* bits are then used to map to a specific node. The node looks at its finger table to try and find the highest node identifier that is less than this filename hash. If there is a node in the finger table that's identifier

is higher than that of the filename, then the file must reside in the predecessor of that node. If the filename hash is still larger than the highest identifier in the finger table, the node queries that node to find the file within its finger table. This iterative process continues until the node that is supposed to have the file is reached. If the file exists, it is returned back to the original caller, otherwise the file doesn't exist within the system, and an error is sent. When adding or removing a file from the system, the same process is repeated as above with the only difference being the operation done on the node.

When a node wants to join the system, it must first connect with another node in the system, and asks what its successor is. The node in the system takes the node identifier of this node, and determines the predecessor of that node, returning the predecessor's successor. This successor then becomes the new node's successor. Using this mechanism, the new node is effectively inserted in between it's predecessor, and that predecessor's successor, which is now the new node's successor. The predecessor still isn't aware of this new node's existence, so a stabilization algorithm that runs periodically is used to update the predecessor's successor to be the new node. This is done by attempting to find the predecessor of a node's successor. If the successor list is up to date, then the predecessor of it's successor should be the node itself. However, if there exists a node in between itself and its successor, it needs to update it's successor to be that node instead. If the successor list was updated, then it notifies the new successor, that it is now it's successor, and that successor updates its predecessor to be the node that notified it. In order to make sure that the finger table is up to date, a fix_finger_table function is also called periodically. In this function, a random index between 1 and B is generated. The first node in the finger table whose value is greater than the finger[i].start, where finger[i].start is $(n + 2^{i-1}) mod (2^m),\ 1 \leq i \leq m$where n is the node identifier of finger[i] is set to finger[i].

Fault Tolerance

In a distributed hash table, a point of failure occurs when nodes leave the system. The files that these nodes hold are lost, and the mapping data associated with these files are now in an inconsistent state. In order to solve these two issues, the distributed hash table uses a stabilization

algorithm to detect node joins and leaves from the network. The mapping information associated with those nodes are updated, and all of the nodes who's mapping information needs to be updated are notified. Files are also replicated on successor nodes on file add, so that no one node leaving the network can destroy the map for the files that it holds and allow the system to lose those files. When a new node joins a system, the files that belong to them are forwarded to them so that if a file search occurs, the file will be found on the correct node.

Performance

A distributed hash table is able to do lookups in O(log *N*) time, where *N* is the number of nodes in the system. This is because the finger table is organized so that each following node in the list is 2^i away from the last, where i is the index in the table. This means that the distance between successors is exponential, allowing the system to skip nodes logarithmically when doing file lookups. Each node is only responsible being up to date on a small subset of the overall network, defined by B nodes, where $N = 2^B$ and N is the total number of nodes in the network. This size defines the size of the finger table and the successor list.

Scalability

A properly developed DHT like Chord is inherently scalable. Distributed systems rely on scalability in order to operate effectively with a significant increase in the number of users of that system [1]. Several approaches can be taken to ensure that scalability will be maintained in a system. These include controlling the cost of physical resources and loss of performance and preventing bottlenecks and the running out of software resources [1]. Our implementation of Chord is able to manage all of these considerations through its strong design.

There are several concepts in the Chord design that assist with scalability. The DHT is able to grow easily with the addition of new nodes [2]. This is critical to allow the system to support larger and larger pools of users. It also allows for users to make a request at any node in the system with the same performance [2]. This reduces a large bottleneck that could have occurred if this aspect of the system had been centralized. Finally, the actual performance of the system grows well with the addition of new nodes. It can be proven that with any number of

nodes in the system N, that the lookup time for any document will run in O(log N) time [2]. Due to this being well below a linear growth rate, it will result in the system always benefiting from adding more nodes.

Consistency

The distributed hash table has two areas where consistency is important. The first is with the mapping table. Since the stabilization algorithm is run periodically, the successor list and finger table will reach equilibrium once the system has stabilized, and will continue to stay up to date. If there are any changes in the system, those changes will also update the successor list and finger table from the stabilization algorithm. The second is with files. In order to keep files replicated, we make sure that when nodes join we give them the files associated with their id, and when nodes leave the successor list, we delete the field associated with them. This way, we make sure that only the successors have the replicated copies of the file.

Concurrency

One of the most important areas that must be managed with Chord is the addition of new nodes. Because the placement and addition of these nodes are not controlled, they must be added in a way that does not disrupt the operation of the system. Instead of trying to ensure that when a node joins the system it is capable of working correctly and at the highest performance, these goals are separated [2]. First, basic correctness is achieved by ensuring that the successor pointer of the new node is correct. Then, incrementally, the finger tables are expanded in order to allow the node to perform well [2]. By separating these concerns, the system is able to guarantee that the addition of any new node does not disrupt any searches that are currently operating, and more importantly that the introduction of many nodes in a short period of time will not disrupt the system as whole. Performance is improved through the process of stabilization that runs periodically and requires no outside management. During this process, existing nodes discover new successors and these successor and predecessor lists are assembled. Similarly, the finger tables are updated in order to hold current information [2].

Although the situation of node joins has been resolved, Chord does not provide any mechanisms for editing files concurrently. Instead an old file must be removed from the system, edited in one place and returned to the system. This mechanism is the responsibility of the application that utilizes Chord to manage.

Major Design Decisions

During the design of our implementation of the Chord DHT, we were presented with many design decisions. For the majority of these decisions, we decided to defer to the specification that was presented in the original paper. This included many of the base algorithms that needed to be implemented and the recursive manner in which requests were made.

However, there were several areas where we decided to tweak the model. The first was with the way in which an application interacts with the DHT. In the paper, the application would be required to first make a request to find the node that is associated with the file that it is searching for and then make a second request to perform an operation such as adding a file. In our design, we decided to combine these operations. When an action like adding, removing, or retrieving a file is performed, the system automatically performs the lookup as well which removes some network overhead. This has the added benefit of making the system less reliant on the user's knowledge of how to interact with a DHT. Instead it can be treated as a black box file storage system. We found that this had the disadvantage of adding additional overhead to the systems processing of a file, but it also allowed us to have stronger control over the replication and fault tolerance of a file. Another area we had to make several design decisions was in the supporting features of this fault tolerance. The original paper offers some solutions for fault tolerance such as successor lists and replication of data, but it does not offer many clear solutions to these problems. Because of this, several additional functions needed to be developed to make sure that replication and fault tolerance were effectively supported by the system.

Related Work

There are several existing structured, peer-to-peer hash tables that are in existence. One of the most well known is Chord, which uses an overlay network that contains a group of successors

spread out across the address space that can be used for node hopping [2]. A future approach, known as Koorde, was later developed that changed the overlay structure to be a de Bruijn graph that allowed it to reach an optimal number of connections between nodes and hops before finding the requested data [3]. Another approach, Kademlia, offers a slightly less rigid approach in terms of required node information and allows for configuration information to be passed through the network at the same time as it is serving requests. It utilizes this by incorporating an XOR metric that operates over a tree structure [4]. Pastry and Tapestry are two other well-known designs that perform similarly to the other network designs that are presented here [5] [6].

Implementation Details

Our implementation relies on several key components to ensure that each node in the system can work effectively.

The first component is the network handler. We took a multi-threaded socket based approach that utilizes thread pools to handle requests. When a request arrives from the network, the main thread of the program un-marshalls the remote function call and places the request into a queue. A pool of worker threads is then responsible for retrieving these requests and performing the required functions. This approach was taken to ensure that the network was not a bottleneck for a node and that many requests could be served concurrently.

The second component is an additional thread that is responsible for certain periodic activities. This thread is responsible for performing the task of stabilization and filling the finger tables. It can be configured to run after sleeping for a certain amount of time. These components were split into this additional thread in order to not disrupt the activities of the network and worker threads.

Both of these components rely on a set of functions that define the underlying actions that a chord node can perform. These are split into two classes. One of these are designated thread functions. These functions define all of the remote calls that another node or entity can call over the network and are connected to the network handler. The rest of the functions define internal functionality that do not require additional support of network connections.

Finally, the bulk of the data has been collected into a class that has been designed to always contain the same data such as the files that the node stores, successor lists, finger tables, and more no matter where or how it is initialized. This decision was made to prevent an object of this class from being passed throughout the program. Instead, when needed, a thread can create a new instance of the class that will contain the exact same inner objects as any other object. This object can also be locked to ensure that the data that is stored within the class does not encounter any concurrency issues within our multi-threaded environment.

We believe that the implementation of the components in this manner leads to a node that performs well under many requests and is flexible enough to be expanded upon and changed to fix issues and support new features.

Demonstration of Example System Run

To run our implementation of the Chord system, several steps need to be taken. First the nodes need to be started. This is accomplished by running the following command with Python 3: `python node.py -p <port> -b <b_choice>` where the *p* parameter is the port that should be assigned to the machine and the *b* parameter determines the size of the Chord ring. Note that the *b* parameter should be chosen the same for all nodes that are on the same ring. Once the nodes have started they can be interacted with through telnet commands that are shown in Appendix A.

When started, each node will display some of their key information periodically. This includes their successor list, predecessor, and finger table. Initially, these values will be set to the identifier that the node itself hashes to because they have not been connected to other nodes. The nodes must be connected using a join command that tells them which node to contact in the current ring. This contacted node will be responsible for finding the new node's position in the ring. Once the nodes join the system, they will begin to update their successors, finger tables, and predecessor.

While nodes are joined and stabilized on the ring, files can be added to the system. Any node in the system can be tasked with adding a file. This node will compute the files identifier and will be responsible for finding and contacting the node that is responsible for holding the file. Nodes that are responsible for holding data will replicate this data over their successors to

protect against fault tolerance. In a similar manner, files can be retrieved and deleted from the system. All of these actions can be performed while new nodes are joining the system and if old nodes have crashed or unexpectedly left the system.

Analysis of Expected Performance

Because of the unique structure of Chord's finger tables, the lookup that underlies the majority of the operations on the Chord system can be performed in O(log *N*) hops from the originator node. This can be intuitively understood by understanding how the finger tables operate. Each finger table contains a series of node references that are offset from the node holding the table. Every successive offset doubles the distance away from the original node that is covered. By looking for the node that is closest to the goal, the distance between the original node and the goal can be cut by half on average which leads to our efficient lookup time.

The distribution of files on the system can also be measured. Because a hash function is used to create the identifiers that the files use, it can be assumed that for any population of files, the identifiers will be uniquely and uniformly distributed across the possible nodes on the ring. Because of this, no single node should be responsible for holding the majority of the files in the system.

Testing Results

We ran a simulator that spins up nodes to a fill factor of 40%, 60%, and 80% using B values of 3, 4, and 5 where B defines the maximum size of the ring, $N = 2^B$. The number of nodes per test can be seen in Table 1.

Table 1: Number of Nodes per Fill Factor.

		Choice of B		
		3	4	5
F. F.	40%	4	7	13
	60%	5	10	20
	80%	7	13	26

Once all nodes join the network and have stabilized, we randomly generate fifty files and add them to the system. Once all files have been added, we search for all of the files and measure the number of hops in the network that each search takes to find the file. We were able to see that the more nodes in the system, on average, the more hops were required to reach each file as seen in Figure 1. We observe that the number of hopes increases, but at a sub linear rate. The average number of hops observed for a system with 7 nodes was 1.43 while the average number of hops for a system with 26 nodes was 2.19. The number of nodes in the system was increased three times, however the average number of hops required to find a file only increased one and a half times. These results are due to the O(log N) lookup times for each file operation.

We also observed that increasing the number of nodes in the system also decreases the number of files that each node has to hold as seen in Figure 2. This natural load balancing feature is due in part to the filenames hash used for lookup, add, and delete operations distributing the files across the namespace in a relatively even manner.

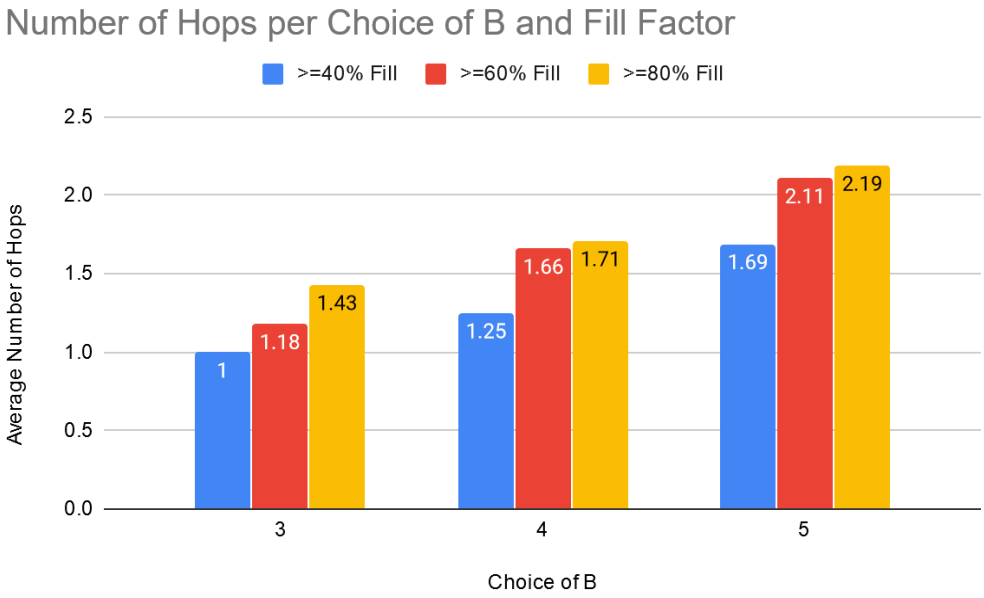

Figure 1: Average Number of Hops for each Choice of B and Fill Factor.

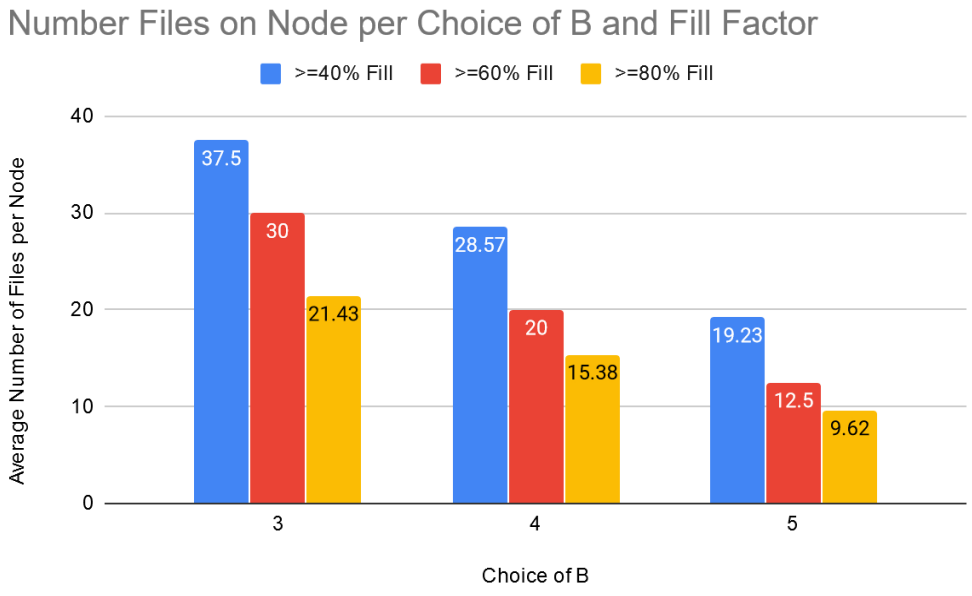

Figure 2: Average Number of Files Present on each Node.

Furthermore, we also see that the standard deviation between the number of files owned by any given node becomes smaller the greater the number of nodes in the network are as seen in Figure 3. This means that the more nodes are added to the system, the more evenly distributed the files are. This also is correlated with the number of hops necessary to locate a file. If the files are distributed on more nodes, the more nodes are in between source and destination nodes, which means that there will be more hops required to find the file.

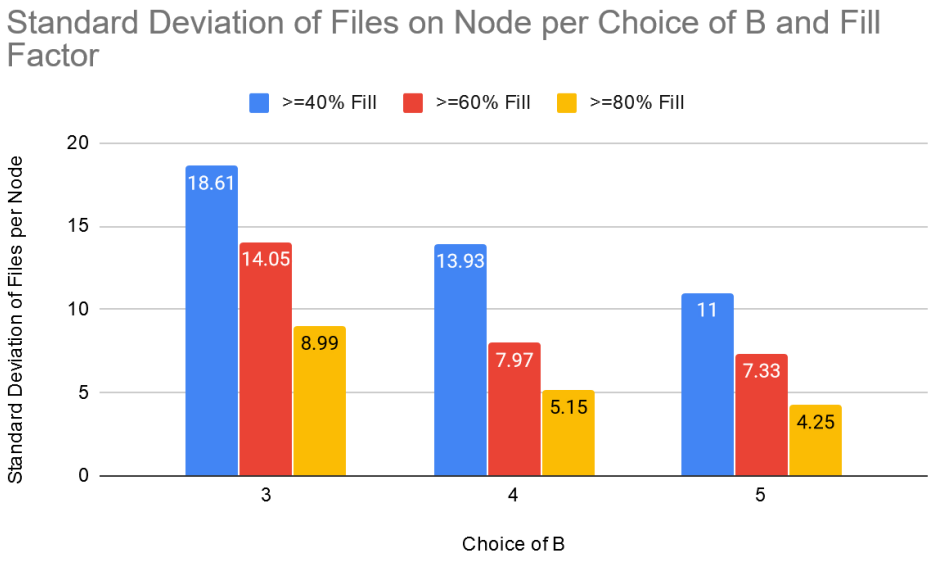

Figure 3: The Standard Deviation of the Average Number of Files Present on each Node.

Conclusion

In conclusion, we successfully were able to implement the Chord DHT which is able to operate robustly and efficiently in a distributed manner. We found that the expected performance that was presented in the original paper was easily reproducible.

If we had more time, we would have spent more time improving the network communication that occurs between our nodes to prevent nodes from communicating with themselves over the network rather than making internal calls. This would have sped up responses and would have reduced the number of threads that would have been used. In addition

we could utilize better techniques to guarantee that the concurrent accesses to internal data structures were better protected with locking strategies. Finally, it would have been rewarding to perform more tests, especially at much larger scales than we were able to perform with our limited resources and time.

Overall, the implementation of the Chord algorithm helped us reinforce skills in the development of distributed systems with regards to fault tolerance, scalability, concurrency, and consistency while maintaining a goal of high performance.

Appendix A: Telnet Commands for the Chord Implementation.

All external commands to the telnet system follow the same general format in order to facilitate remote execution. They each contain an action which corresponds to an internal function and a list of parameters that can be full or empty based on the specific action assembled into JSON. The number of characters that this string contains is prepended onto the string and separated from the command with a tilde (~). Here are all of the public API calls that can be made to the system:

Join command

The join command tells which node can be contacted that is already present in the ring and which can find the location that the calling node will reside.

Action: `join`

Parameters: `[<IP of node already in ring>, <port of node already in ring>]`

Example: `49~{"action": "join", "params": ["127.0.0.1", 5000]}`

Add file command

The add file command tells a node to add the file to the system by finding the node that the file should reside on. Note, true is always required in the command.

Action: `add_file`

Parameters: `[<filename>, <file contents>, true]`

Example: `60~{"action": "add_file", "params": ["file.txt", "text", true]}`

Get file command

The get file command finds the node that should currently hold the file and returns the data that node holds. Note, true is always required in the command.

Action: `get_file`

Parameters: `[<filename>, true]`

Example: `52~{"action": "get_file", "params": ["file.txt", true]}`

Get files command

The get files command returns all of the files that a node currently holds.

Action: `get_files`

Parameters: `[]`

Example: `37~{"action": "get_files", "params": []}`

Remove file command

The remove file command finds the node that should currently hold the file and removes it. Note, true is always required in this command.

Action: `remove_file`

Parameters: `[]`

Example: `55~{"action": "remove_file", "params": ["file.txt", true]}`

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, "Distributed Systems: Concepts and Designs", Addison-Wesley, 2012

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", MIT Laboratory for Computer Science, Cambridge, MA Link

[3] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table", MIT Laboratory for Computer Science, Cambridge, MA Link

[4] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", New York University, New York, NY Link

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer system", Microsoft Research, Cambridge, UK Link

[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment", IEEE Journal on Selected Areas in Communications, January 2004 Link