Incremental Chord Protocol

Christer Hagenes Opdahl
Kim Grønning Eide

September 2020

Contents

1 Introduction

This paper will describe how to implement a peer to peer network based on the "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications" [2]. The peer to peer network functions as a distributed system which is organized without any master node that controls the network. Each node has equal control networks functions.

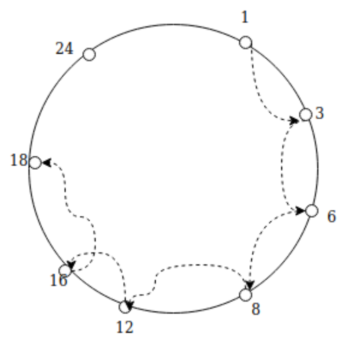

Figure 1: Shows the lookup for a circle in a incremental manner.

2 Background

The chord protocol utilizes distributed hash tables, which maps key-value pairs into buckets[1], this makes Chord a fast lookup protocol where every node becomes a bucket for the key-value pairs. If a key maps directly to a nodes identity given the key space, then that node should keep the key in its value store. If that node does not exist, then the key should be stored in the node which immediately succeed the key identity. Figure 1 shows how a request is forwarded through the chord circle. If node 1 were given a request to get a key which mapped to node 17, then we would have to traverse the chord circle incrementally until the successor of node 17 was found, which in figure 1 is the node with identity 18. Each identity is given to the node by hashing the node name or its IP address with sha1 or equivalent hash functions. In addition our solution chooses to do a modulo operation on the identity making debugging and visualisation easier. This however is not needed for the chord protocol to work.

$$Identity = hashed(name)\ modulo\ m$$

where m is the maximum number of nodes within the chord circle. This way of traversing the chord circle makes the lookup times equivalent to O(n), more advanced solutions makes the lookup times faster by implementing finger tables.

3 Related Work

When comparing our work with others, we assume a chord-based application that utilizes a key-value store to map the values to a node. The key has to be a hashed value, or a string that is able to be hashed. Our implementation is based on the "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications" [2] paper. The work they have described uses a finger table to lookup the keys while our version uses an incremental approach where the request is sent to the successor. Each node handles values that are mapped from numbers ranging from their own ID until their predecessors ID.

4 Description of System

The network is initialized with just a single node, that gets its port number from an argument given when started. As new nodes is to join the network they start a server of their own, and sends a get request to the other nodes to receive their neighbors. The node then calculates its own ID, which in turn gives the node its place on the chord ring. When the node has found its own place, it sends a notify message to its new neighbors that it has arrived, and that they should update their successor and predecessor accordingly. A node that want to join the network, can request to join from any existing node on the network.

When the network receives a PUT request from a client, the node that receives the requests checks if it should store the key in its own value store. To figure this out, the key is hashed with a sha1 protocol, and then mod M on the hash value. If the hashed value is not a value between the current node and its predecessor, the request is sent to the successor to find a better fit. When the node that has the hashed value ranging from between itself and its predecessor, it stores the value in a dictionary on the key, and status code 200 is sent back to the client.

The GET requests is handled in a similar manner. Where the node that receives the requests checks if the hashed key value is between its own ID and the predecessors. If it is, and the key is not stored at the current node, status code 404 is sent back to the client. If its not within the correct range the request is forwarded to the node in the system that should hold the key. When the correct node is found the response is done in a reverse order as the forwarding where done. So the same node that received the request is also the node that responds to the client with either the value or not found code.

5 Evaluation

When testing the performance of the network, we utilized a client that measured the time spent for first storing a key-value pair, and then trying to retrieve the same key-value pair from the network. All times used in the plots are times from testing the chord network on a local cluster. In the test results listed below, figure 2 plots the different times for 1000 put request and then 1000 get request for differing number of nodes. The orange line is the plotted times for retrieving the same key-value pair but on different nodes than it was originally put on, the blue line are times for retrieving from the same node that the put request was sent to. As you can see, the times are almost equal in comparison. This shows us that the chord protocol works as intended, because the node that receives the put request is not necessarily the same node that the key-value pair should be stored on, and we should expect the times to be somewhat equal. In addition we should expect to see a linear time increase when increasing the number of nodes due to the nature of the incremental lookup.

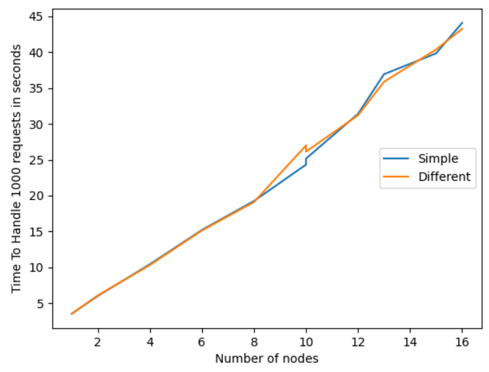

Figure 2: Shows the time it takes to process 2000 request for different number of nodes

Figure 3 shows the calculated throughput.

$$throughput = Numberofrequest/totaltime$$

We see that as the chord circle grows, the number of request handled per second decreases as a result of an increase in communication between the nodes to find the correct space to store and retrieve keys. This decrease can be diminished by implementing finger tables as described in "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications" [2]

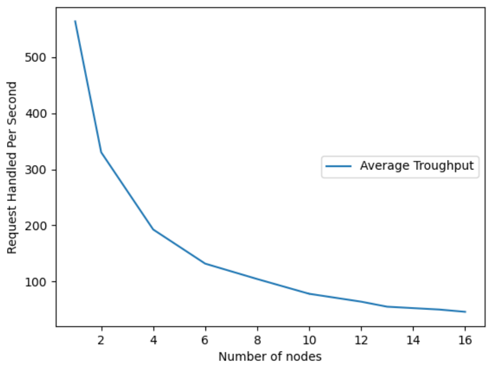

Figure 3: Shows the average throughput for different number of nodes in the network

6 Future Work

When there is a new node that joins the network, the node just joins without asking its successor to receive the values it should hold. This could lead to issues later on when asking for the given key, and the wrong node is asked for it. To avoid this in the current version, no nodes are to be added in the chord circle after key-value pairs are stored. This is of course neither future proof or scalable, but the implementation of finger tables are planned in the future.

To implement finger tables, each node would get its own table which holds the nodes of $id + 2^n$. Then the node can jump to the highest node in the finger table, that holds an ID lower than the hash value of the key. There can be an issue in this were if the key has a lower hash value than the node. Then the key should be sent to the highest value so it can take a full circle in the chord ring and start from the first available node.

7 Conclusion

This paper describes how we implemented a version of the chord protocol which supports functionalities such as incremental lookup, node joining with notification of predecessor and successor.

References

[1] Maarten Van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Pearson Education, Inc., London, England, 3rd edition, 2017.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocolfor internet applications.