Key-value Store

INF-3200 - Distributed Systems Fundamentals

Tommy H. Pettersen
Thomas Luzi

September 28, 2020

Contains 5 pages

Contents

1 Introduction

This paper describes the implementation of a Chord-inspired approach for storing key-value pairs in a decentralized distributed system. It describes how to set up a basic system that has $O(n)$ speed regarding lookup, storage and membership changes and looks at some of the key aspects of a distributed hash table.

2 Technical background

2.1 Key-Value Store

A key-value store is simply a storage of values mapped to keys. The value can be any type of data, depending on what the service provides. A key-value store is the basis of both distributed hash tables[2] and the Chord protocol[1].

2.2 Distributed Hash Tables

A distributed hash table (DHT) is a decentralized storage system that provides lookup and storage schemes similar to a hash table, storing key-value pairs.
Each node in a DHT is responsible for keys along with the mapped values. Any node can efficiently retrieve the value associated with a given key.[2]

2.3 Chord Protocol

Chord is a distributed lookup protocol that provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps.[1]

3 Related Work

This paper describes an implementation inspired by Chord protocol, but the two protocols still differs in several ways. The most distinct difference between the two, is the lack of finger-tables for this implementation, of which is one of the key-aspects that lets Chord achieve its $O(\log_2 n)$ speed regarding lookup, storage and membership changes. Because of the lack of finger-tables, the lookup and storage speed for this implementation is limited to $O(n)$. While a Chord protocol will always travel clockwise when traversing the network, this protocol will travel clockwise/counter-clockwise depending on the hash-value of a given key compared to a node's own hash-value. Also, nodes in our implementation does not have an uniquely assigned id, but rather use their hashed IP-address to locate their respective place in the network and which items they should store.

4 Design

Being that this is a protocol used in decentralized peer-to-peer systems, this protocol allows any given peer to start its own network or to join an already existing network.

4.1 Creating/Joining Network

When a node is initialized without an address to connect to, it hashes its own IP-address and port. It then listens for any incoming requests on its respective port. When a node is initialized with an additional server-address to connect to, it will send a http request, asking to join the respective network. The receiver of the message will interpret the message to be a join request. It will then hash the address of the sender and look for its respective place in the network. The hashed value will act as an identifier for that respective node in the network, and nodes

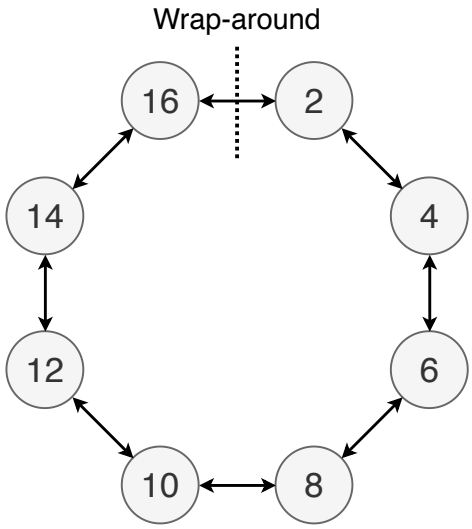

Figure 1: The peer-to-peer network, where each number represents a key

are stored in a ring in ascending order, starting from low to high, then wrapping around (see figure 1). The request is routed across the network until the request reaches one of its future adjacent nodes. The appropriate node will update both its own successor/predecessor, and also the corresponding adjacent node's successor/predecessor. Lastly it will send a response back to the newly joined node with its appropriate successor and predecessor. This process is done recursively and the response will be routed using the same path as the original request. The newly joined node receives the response and updates its successor and predecessor accordingly, thus joining the Peer-to-peer network.

4.2 Traversing Across the Network

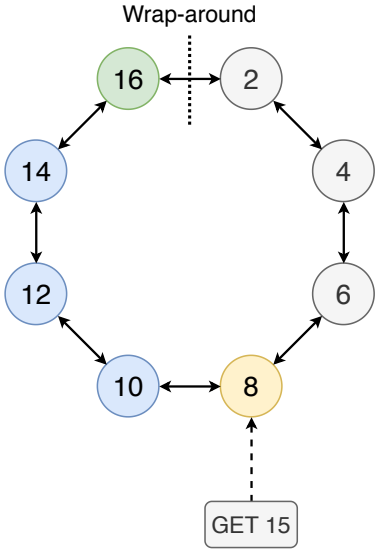

Figure 2: Network routing when key is greater than the entry node

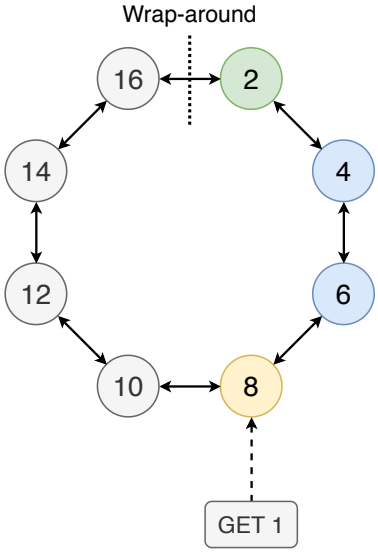

Figure 3: Network routing when key is less than the entry node

The traversal mechanism for this protocol is implemented to handle routing both clockwise and counter-clockwise, depending on a given key relative to a respective node's own key. As illustrated in figure 2, node $n = 8$ receives a GET request on key $k = 15$. The respective node compares

whether the received key is greater or less than its own key. Since the requested key is greater than its own key, the request is routed to node $n = 8$'s successor, node $n = 10$. Node $n = 10$ does the same comparison and routes the request to its successor, node $n = 12$. This process continues until the request reaches node $n = 16$. This node determines that it should be the owner of that key-value pair, if it exists. If node $n = 16$ locates the stored value that maps to the given key, and sends the respective value in response. The response is routed through the same path as the original request and returned to the client. Figure 3 illustrates the same scenario, only that the request is routed counter-clockwise.

4.3 Storage/Retrieval of Key-Value Pairs

A node will only store key-value pairs where the keys are between its own key and its predecessor's key. E.g if we look at figure 1, node $n = 8$ will store key-value pairs with keys 6 and 7. This also applies for the node with the lowest key, E.g node $n = 2$ will store key-value pairs with keys greater or equal to 16, and keys less than 2.
Since the storage distribution is based on the cryptographic hash function SHA1, which is a consistent hash function, the key-value pairs are relatively load balanced throughout the network if the number of nodes in the network reaches an adequate size.

5 Discussion

There is a problem that this implementation does not handle. If a new node joins a certain network that already stores several key-value pairs, there is a fair possibility that some of the key-value pairs now maps to this respective node and that those objects should be transferred to their rightful owner. The current problem is that a following GET request for one of these respective key-value pairs will route to the new node. Since the respective node does not have the data, the request will fail.
Another issue that this protocol does not address is the notion of nodes leaving the server, either voluntary or involuntary(failures etc.). If a single node fails, the whole network is exposed to failure.
Since our implementation handles lookups both clockwise and counter-clockwise, the speed of a lookup process is on average faster than just iterating one direction, even though the worst-case speed will remain the same.

5.1 Experiments

To test the performance of the implemented protocol, a client was set up to do 40000 request to an established network, where half of them were PUT requests and the other half were GET requests.
As can be seen in figure 4, as the number of nodes in the network grows, the throughput drastically drops. The reason for this behaviour is because as the number of nodes grow, each request may need to route through the network to be resolved. As each node in the network only keeps track of its immediate neighbors, this routing is an $O(n)$ operation.

5.2 Future Work

Our current solution does not support key relocation and this seems like the most natural place to further develop this protocol. This improvement could implemented by having the successor of a newly joined node check whether some of its stored key-value pairs should be transferred to the new node.
In order to fix the issue regarding nodes suddenly leaving the network, the nodes should periodically send messages to their neighbors to check if they are still alive. If one of those messages would fail, the affected node should send a request the other direction asking for a node that is also missing a neighbor, and from there they could link up, thus re-establishing the connection. This would not be optimal if two non-adjacent neighbors left the network simultaneously as this would in effect create two separate networks.

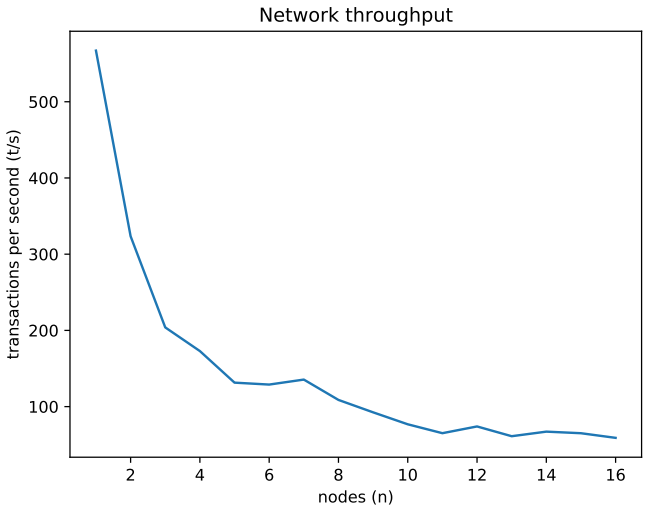

Figure 4: Transaction per second as the number of nodes in the network grow

In order to achieve greater lookup speed, a finger table could be implemented. This would improve the lookup, storage and membership changes drastically.

6 Summary

This paper described the implementation of a simplistic Chord-inspired approach for storing key-value pairs in a decentralized distributed system. It described how to set up a basic system that has $O(n)$ speed regarding lookup, storage and membership changes and looked at some of the problems regarding transactions as the number of nodes in the network grows, and the problems due to a node leaving the network.

References

[1] Ion Stoica et al. “Chord: A scalable peer-to-peer lookup protocol for Internet applications”. In: *IEEE Transactions on Networking* 11 (Feb. 2003). DOI: 10.1109/TNET.2002.808407.

[2] *What is a distributed hash table?* URL: https://www.educative.io/edpresso/what-is-a-distributed-hash-table.