# COSC 373 Final Project: Distributed Storage

Dhyey Mavani, Matthew Kaneb, Sun Ho Yoo

5/12/2022

## Overview

The primary research question addressed in this paper is how one would store 1 M movies, each with a size of about 1 GB, on 1 M nodes, each equipped with a 1 TB disk. Additional concerns include dealing with nodes and information entering and leaving the network. As the amount of data involved in every aspect of our lives continues to increase, storing it all in one place is not flexible or fast enough and often can be prohibitively expensive. Furthermore, a centralized approach presents a single point of failure that can be catastrophic when the data isn't movies but instead medical records or financial data. Companies like Amazon, Facebook, and Google have already adopted distributed storage technologies and have incorporated them into products you already use every day. While our research question focuses on the storage of movies, the solution to this problem can be applied to any number of important settings.

To answer our question, we begin by explaining consistent hashing, a distributed hashing scheme that answers the question of what is stored where and how queries will work as well as how will the network handle nodes and information entering and exiting. Next, we will discuss a few simple network structures. Finally, we will combine these two into a general distributed hash table ($DHT$) algorithm to answer our original research question.

## Consistent Hashing

**Algorithm 21.1** Consistent Hashing

1: Hash the unique file name of each movie $m$ with a known set of hash functions $h_i(m) \to [0, 1)$, for $i = 1, \ldots, k$
2: Hash the unique name (e.g., IP address and port number) of each node with the same set of hash functions $h_i$, for $i = 1, \ldots, k$
3: Store a copy of a movie $x$ on node $u$ if $h_i(x) \approx h_i(u)$, for any $i$. More formally, store movie $x$ on node $u$ if

$$|h_i(x) - h_i(u)| = \min_m\{|h_i(m) - h_i(u)|\}, \text{ for any } i$$

Figure 1: Consistent Hashing Algorithm Pseudocode

Consistent hashing is a distributed hashing scheme that makes considerable progress in answering our original research question by figuring out which nodes will store which movies and how they can be accessed. The algorithm uses shared hash functions to decide which movies are stored where and then uses those same hash functions to help direct queries toward the node where the requested movie is located. The algorithm also elegantly handles nodes entering/exiting the network.

Because the hash functions are applied to both the nodes and movies, they are mapped to the same space, meaning the algorithm operates independently of the number of nodes or movies. This means that when a node leaves the network, only the movies mapped to that node will be re-hashed, not the entire set of movies in the network. Furthermore, when a node arrives into the system, this same rule will re-assign $\frac{m}{n}$ movies to be stored on the new node.

| | Classic hash table | Consistent hashing |
|---|---|---|
| add a node | O(K) | O(K/logN) |
| remove a node | O(K) | o(K/logN) |
| add a key | O(1) | O(logN) |
| remove a key | O(1) | O(logN) |

Figure 2: Consistent Hashing Complexity

While it is possible that some movie does not hash closest to a node for any of its hash functions, this is highly unlikely: For each node $(n)$ , each movie has about the same probability $(\frac{1}{m})$ to be stored. By linearity of expectation, a movie is stored n/m times, in expectation. Having each node store the same amount of information in expectation is a desirable result with respect to the central point of failure problem presented by centralized storage paradigms.

To see more concretely how this algorithm works, let us see an example.

**Consistent Hashing Example**

$$\text{Movies} = \{\text{Matrix, Shrek, Batman, Spiderman, Mad Max }\}$$

$$\text{Nodes} = \{\text{A,B,C}\}$$

Given the movies above, the first step of the consistent hashing algorithm is to hash the unique file name of each movie $m$ with a known set of hash functions $h_i(m) \rightarrow [0, 1)$. In this simple example, there is only one hash function $h(k)$ which takes strings and maps them to $[0, 1)$. The resulting hash values are displayed below.

| **Step 1: Hash Movies** | |
|---|---|
| Key | h(Key) |
| Matrix | 0.163 |
| Shrek | 0.759 |
| Batman | 0.500 |
| Spiderman | 0.979 |
| Mad Max | 0.342 |

The next step of the algorithm is to hash the node IDs in a similar way to the first step, using our one hash function $h(k)$. The results of this step are displayed below.

| **Step 2: Hash Nodes** | |
|---|---|
| Key | h(Key) |
| A | 0.557 |
| B | 0.808 |
| C | 0.227 |

The final step is to use the above values to store movies at nodes that satisfy the following equation: $|h(x)-h(u)| = \min_m\{|h(m)-h(u)|\}$. To find where "The Matrix" should be stored, the following calculations would be made:

**- Should "The Matrix" be stored on node A?**

$$|h(\text{The Matrix}) - h(A)| = |0.163 - 0.557| = 0.394$$
$$\min_m\{|h(m) - h(u)|\} = \min_m\{|h(m) - 0.557|\} = \min[0.394, 0.202, 0.057, 0.421, 0.215] = 0.057$$
$$|h(\text{The Matrix}) - h(A)| \neq \min_m\{|h(m) - h(A)|\} \text{ (NO)}$$

**- Should "The Matrix" be stored on node B?**

$$|h(\text{The Matrix}) - h(B)| = |0.163 - 0.808| = 0.645$$
$$\min_m\{|h(m) - h(u)|\} = \min_m\{|h(m) - 0.808|\} = \min[0.644, 0.048, 0.308, 0.171, 0.466] = 0.048$$
$$|h(\text{The Matrix}) - h(B)| \neq \min_m\{|h(m) - h(B)|\} \text{ (NO)}$$

**- Should "The Matrix" be stored on node C?**

$$|h(\text{The Matrix}) - h(C)| = |0.163 - 0.227| = 0.064$$
$$\min_m\{|h(m) - h(u)|\} = \min_m\{|h(m) - 0.227|\} = \min[0.064, 0.532, 0.273, 0.752, 0.115] = 0.064$$
$$|h(\text{The Matrix}) - h(C)| = \min_m\{|h(m) - h(C)|\} \text{ (YES)}$$

Because the condition was only met for node $C$, "The Matrix" will only be stored at $C$. These calculations would be repeated for each combination of movie and node.

Consistent hashing is typically visualized using circular graphics like the one below. This is because the seemingly abstract rule used above to determine which nodes should store a movie becomes much more intuitive when expressed in circular form. It is easy to see that the above rule corresponds to having a movie stored at its nearest counter-clockwise neighbor, resulting in the following:

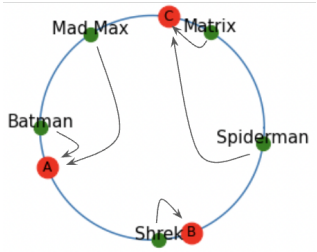


Figure 3: Consistent Hashing Visualization

Once it has been determined which nodes should store which movies, other nodes (with access to the same nodes and hash function) can use the steps above to figure out which node has the movie they desire.

A key characteristic of this algorithm of hashing is its robustness to nodes and movies leaving the network. Suppose $B$ leaves the network. Instead of re-hashing all of the movies in the network, only the movie "Shrek" is re-hashed using the process described in Step 3 and is now stored on node $C$, resulting in the following:

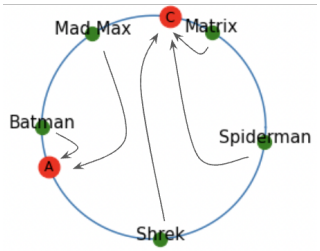


Figure 4: Consistent Hashing with Churn

With the storage and querying of the movies taken care of by consistent hashing, the question remains how the network will be structured. To answer this, we must dive deeper into network structures.

# Network Structure

So far, we have discussed a scheme which allows for the storage and querying of movies in some network of nodes. However, it is impractical to assume every node will be connected to every other node and be able to ask for data explicitly from the node that stores it. Thus, the key practical consideration that remains to be addressed is how the network will be structured. Different topologies have different benefits and shortcomings for various use cases. But before diving into some common network structure options, we will first discuss some desirable qualities of any network structure.

- **Unique IDs**: In order for consistent hashing or any hashing scheme to work, the nodes in the network need a unique way to refer to them.
- **Homogeneity**: The network should be as homogeneous as possible. This way no node can bottleneck performance or be a single point of failure.
- **Minimal Degree**: Every node should have a small degree. This will allow every node to maintain a persistent connection with each neighbor, which will help us to deal with churn.
- **Minimal Diameter**: The network should have a small diameter. This makes routing easier because when a node does not have information about a data item, then it can get it from neighbors in as few steps as possible.

## Fat Tree Structure

Trees are a common choice of network structure because they make routing very easy. This advantage comes from the fact that for every source-destination pair there is only one path. However, since the root of a tree is a bottleneck, trees are not homogeneous. Instead, so-called fat trees should be used. Fat trees have the property that every edge connecting a node $v$ to its parent $u$ has a capacity that is proportional to the number of leaves of the sub-tree rooted at $v$. This way, each layer has the same aggregate bandwidth.

Fat trees have many desirable qualities like fast and simple routing and, when combined with consistent hashing, are able to solve our initial research question. However, they have a few shortcomings. Firstly, fat trees are not homogeneous, meaning some of the nodes carry far more of the network load than others. This means the network may be susceptible to bottlenecks and single points of failure. Furthermore, in practice

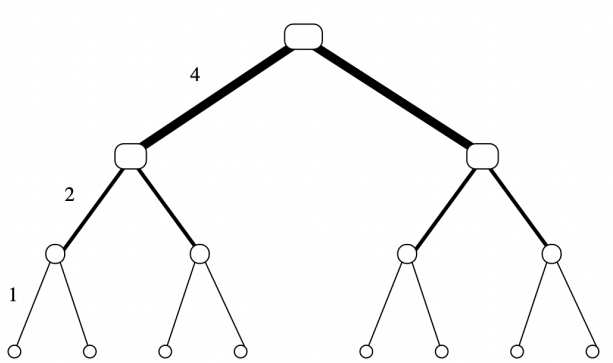


Figure 5: Fat Tree Example

| Diameter | Edges | Degree |
|---|---|---|
| O(N) | O(N) | 2 |

Figure 6: Fat Tree Complexity

it is easier to build a network of nodes that are all the same than it is to build one where at every level of the tree, the nodes are different. In the next section we discuss the butterfly network, which addresses the shortcomings of the fat tree without giving up efficiency.

## Butterfly Network Structure

A butterfly network is another way to structure nodes into an efficient network with many of the desirable properties we hoped to have. More formally, a $d$-dimensional butterfly $BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 + E_2$ where $E_1$ and $E_2$ are defined below:

$$E_1 = \{\{(i, \alpha), (i+1, \alpha)\} | i \in [d], \alpha \in [2^d]\}$$
$$E_2 = \{\{(i, \alpha), (i+1, \beta)\} | i \in [d], \alpha, \beta \in [2^d], |\alpha - \beta| = 2^i\}$$

Below is an example of a 3-dimensional butterfly $BF(3)$.

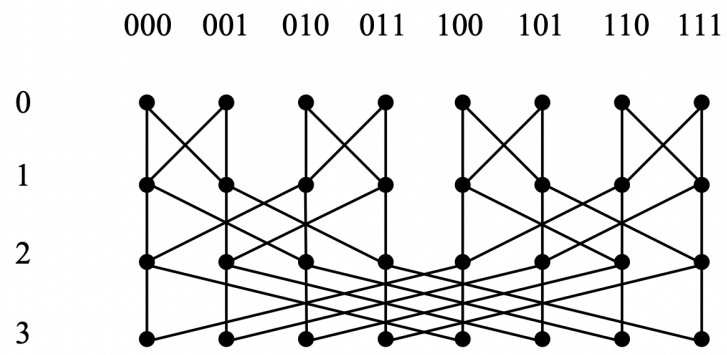


Figure 7: 3-Dimensional Butterfly Network Example

In general, $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^2$ edges, and a degree of 4. Fat tree structures and butterflies are actually quite similar. By merging the $2^i$ nodes on level $i$, the butterfly network becomes a fat tree.

| Diameter | Edges | Degree |
|---|---|---|
| O(N) | O(N) | 4 |

Figure 8: Butterfly Complexity

**Routing in a Butterfly Network**

Routing in butterfly networks is made easy by using the binary representation of node IDs to inform routing decisions. For every bit in the ID, 0 is interpreted as routing down the left sub-tree and 1 down the right. See below how two nodes use this routing scheme to reach the node with ID 011.

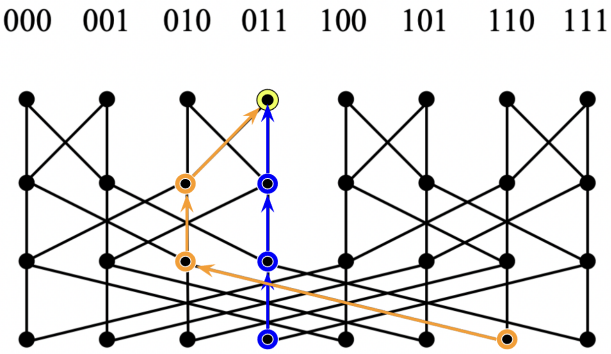


Figure 9: Butterfly Network Routing Example

Butterfly networks take the fast and simple lookup of fat trees and fix the problem of homogeneity, alleviating the risk of bottlenecks and single points of failure. While there are many choices of network structure, butterfly networks and extensions of it are a common choice of tree-based structure. Extensions of butterfly networks like Benes networks and Clos networks are regularly used in telecommunication and data centers to solve the research question we aim to address today.

# Distributed Hash Tables (DHTs)

A distributed hash table (DHT) is a distributed data structure that implements a distributed storage using some hashing scheme like consistent hashing paired with some network structure like a fat tree or butterfly network. A DHT should support at least (i) a search (for a key) and (ii) an insert (key, object) operation, possibly also (iii) a delete (key) operation. Let us consider how each of these operations would work in a DHT using consistent hashing and a butterfly network structure.

- **Searching for a Movie**: To retrieve a movie from the DHT, first you must use the universally known hash functions to hash the movie and find out the ID of the node(s) it is stored on. Next, using the bits of the node's ID your order will be routed by the network to the node with your movie stored on it and it will be sent back to you along the same path it came on.
- **Inserting a Movie**: To insert a movie, first hash it and then send it to the node(s) that the hash function(s) tell you it should be stored. Now, users can query this movie and it will be stored at the node it is expected to be on.
- **Removing a Movie**: Removing a movie is trivial. Once deleting it from the node it is stored on, anyone who asks for it will reach that node and it will inform them that it is no longer available.

The process discussed in this paper also account for churn. Let us consider what a node entering or exiting the network looks like in a DHT using consistent hashing and a butterfly network structure.

- **New Node Joins the Network**: Suppose a node joins the network. Adjacent nodes will re-hash their movies and send the ones that should be stored on the new node to that node.
- **Node Leaves the Network**: When a node leaves the network, the movies that hash to it are re-hashed and stored on adjacent servers based on the results of the hash function(s).

## Conclusion

In this paper, we set out to address the problem of distributed storage. How do you store $1M$ movies, each with a size of about 1 GB, on 1 M nodes, each equipped with a 1 TB disk? To answer this question, we first discussed consistent hashing. This distributed hashing scheme does much of the work addressing the problem of distributed storage. Using consistent hashing, a set of nodes can add, delete, and query movies in a distributed manner. Furthermore, consistent hashing is robust to nodes entering and leaving the network. However, consistent hashing only provides movie watchers with the ID of the node where their movie is stored. The way they contact and retrieve the movie from that node is dependent on the structure of the network. In this paper, we discussed two network topographies, fat trees and butterfly networks, which aim to minimize degree and diameter for efficient routing as well as disperse the load any one node carries. The combination of distributed hashing scheme and network topography create a distributed hash table (DHT) algorithm which supports searching for a key, inserting key-value pairs, removing them, and entering/exiting nodes, thus answering our initial research question.

## Future Work

While we have provided a brief overview of distributed storage, there is much more to consider before declaring this problem to be solved. Below we have included a few areas ripe for further inspection.

- **Distributed Hashing Schemes**: In this paper, the only distributed hashing scheme discussed was consistent hashing. While it is very popular, alternatives exist. The other main distributed hashing scheme not discussed in this paper is rendezvous hashing. Further work could be done exploring the benefits and disadvantages of either hashing scheme for various use cases.
- **Network Structures**: In this paper, the two network structures we focused on were fat trees and butterfly networks. However, many more network structure options exist including various extensions of trees, rings, grids and tori. Future work could include further study of these network topography choices.
- **Hypernodes**: In high churn environments, when nodes are constantly entering and dropping out of the network, nodes will only be in contact with a small number of neighbors. No single node can have an accurate picture of what other nodes are currently in the system. Instead, each node will just know about a small subset of 100 or less other nodes ("neighbors"). This way, nodes can withstand high churn situations. In this case, this group of 'neighbors' will act as a hypernode in a network of other hypernodes, instead of each node acting individually. Further work could be done to look more into how these hypernodes and formed, arranged, and maintained as nodes come and go.
- **Security**: While we consistently address the concern of having a single point of failure, there are a myriad of other attacks that one could unleash onto a network. Future work could explore how distributed hash tables in practice combat adversarial activity.

# Bibliography

Carzolio, Juan Pablo. “A Guide To Consistent Hashing”. Toptal Engineering Blog, 2017, https://www.toptal.com/big-data/consistent-hashing.

Karger, David et al. “Web Caching With Consistent Hashing”. Computer Networks, vol 31, no. 11-16, 1999, pp. 1203-1213. Elsevier BV, https://doi.org/10.1016/s1389-1286(99)00055-9.

LeBlanc, Thomas J. et al. “Large-Scale Parallel Programming: Experience With BBN Butterfly Parallel Processor”. ACM SIGPLAN Notices, vol 23, no. 9, 1988, pp. 161-172. Association For Computing Machinery (ACM), https://doi.org/10.1145/62116.62131.

Wattenhofer, Roger. “Principles of Distributed Computing”. Zurich, Swiss Federal Institute of Technology, 2016.