SANTA CLARA UNIVERSITY

THE JESUIT UNIVERSITY IN SILICON VALLEY

Hash Algorithm evaluation In Distributed File System

Team: open-minded

Yuan Song
Jing Jin
Yiwen Mo

Abstract

A key feature of a distributed file system is to distribute data to different data nodes, where these data are actually stored. To develop that feature, generally, hash algorithm is a feasible choice. But in a distributed file system, the issue is that because of the huge scale of data nodes, the possibility of data nodes change becomes too big to be ignored. That causes traditional hash algorithm cannot competent for this work. So a new hashing algorithm, Consistent hashing, become one of the key technique that has been explored to solve this issue.

In this project, we will implement these two algorithms, traditional hash and Consistent hash, and a distributed file system simulator and evaluate these two algorithm quantitatively in that simulator in terms of performance and availability.

Keywords:
Consistent Hashing, distributed file system, distributed file system

Summary

One aspect of a distributed file system is to map data to different data nodes. In order to explore this aspect, we research on the hash algorithm. However, the problem is that due to hundred and thousand of data nodes in a distributed file system, data node change becomes normal cases. That causes the traditional hash algorithm cannot compete for this work. So a better solution, consistent hashing, becomes one of the critical technique that has been explored to solve the problem.

In this project, we implement these two algorithms, traditional hash and Consistent hash, and a distributed file system simulator and evaluate these two algorithms quantitatively in that simulator in terms of performance and availability.

We evaluate the performance of two algorithms in the simulated environment in terms of balance and smoothness these two indexes. The balance property is measured as the standard deviation of number of files allocated on each node. In general, as number of data nodes increases, the balance of two algorithms is both improved. Both algorithm exhibits well-balanced property, especially for large number of data nodes. Overall, there is no obvious difference between these two algorithms in terms of balance.

The smoothness property is measured as the number of chunks that would be relocated to another data node, when adding a new data node into system and a data node crashing. According to the result of testing, when number of data nodes increase to 512, almost all the chunks in the file system need to relocate for traditional hashing algorithm. This would be a disaster for causing huge overhead when a new data node is added. On the contrary, only a small part of chunks need to relocate for consistent hashing algorithm and that part of chunks becomes trivial when number of data nodes increases to 512 because the intervals on the circle corresponding to that part of files becomes smaller.

In addition, this project evaluates the availability of the whole system, which is recover time needed to relocate the chunks. The results are similar to that of smoothness. For traditional hashing, recover time increases significantly until the number of data nodes reaches to some point that almost all the files need to relocate, and the availability cannot satisfy the user when data nodes changes a lot. However, the recover time decreases as the number of data nodes increases.

To sum up, based on the result of our simulation, these two algorithms both could satisfy the balance of allocating files for large number of data nodes. More importantly, file system using consistent hashing algorithm would be better in terms of smoothness and availability, especially for large number of data nodes. In this case, we could conclude that consistent hashing algorithm is more suitable for distributed file systems.

Contents

I. Introduction

A key feature of a distributed file system is to route data to different data nodes, where these data are stored. One kind of technique mainly applied in this field is hashing technique. But, unfortunately, traditional hashing has fatal defect in this situation.

As we all known, traditional hashing can efficiently distribute and retrieve data to multiple, even very large amount of, nodes of the hashing system. But in distributed environment, a situation, which happens so much frequently that cannot be ignored, is that the nodes in the hashing system are not that stable. It will be planed or involuntarily fail. The nodes are possibly be updated, replaced or added according to plan. On the other hand, the nodes or disks in the system may crash because of hardware error. While traditional hashing distributes data based on the amount of the nodes in the system. That means all data in buckets have to be relocated if any node change, no matter the number of nodes increasing or decreasing. Therefore, traditional hashing might not be efficient on that situation. It is necessary to find a new method to solve this problem. Consistent hash algorithm might be the answer.

II. Literature Review

1. Classical hashing vs. consistent hashing algorithm

Distributed hash table (DHT) is a fundamental algorithm used in distributed systems. It uses buckets to store the key-value pair similar to the hash tables, and each bucket can reside in a separate node or machine. And any participating nodes can efficiently retrieve the value stored in the buckets with the hashing function. Research on DHT is largely motivated by the application in peer-to-peer system because the resources are distributed across the Internet. It is also used in web caching, distributed file systems, domain name services, etc.

A simple scheme for implementing DHT is using modulus operation on key. This implementation is simple and feasible in some circumstances. However, it does have a significant drawback that it fails when nodes are added to or removed from the system. When the nodes change, nearly every object will be hashed to a new location and the systems may fails unless all the data are copied to the proper buckets, which may cause extremely high load for the system. Therefore, new algorithms for reducing moving of data are needed to solve this problem.

Consistent hashing is a special kind of hashing algorithm that was first introduced by Karger, Lehman, et al. [1] in order to develop caching protocols that do not require users to have current or consistent view of network. It reduces the changes as the range of hash function changes, and also applies to the implementation of DHT. Consistent hashing algorithm defines a view to be the set of caches of a particular client is aware of. Each machine is aware of a constant fraction of the currently operating caches. When a machine is added to or removed from the set

of caches, the expected fraction of objects that must be moved to a new cache is the minimum needed to maintain a balanced load across the caches.

Karger, Sherman, et al. [2] provide a simple-to-implement consistent hashing function that satisfies the needs. First choose some standard hashing functions to map objects to the range [0,1], which can be thought as the unit circle. Thus both objects and nodes can be mapped as points on the same circle. And every object is assigned to the first cache whose point it encounters moving clockwise from the object's point. As a result, each bucket has all the objects mapped between itself and the previous bucket. If a node is removed or unavailable, the resources this bucket holds will be redistributed to remaining buckets, while the resources in other buckets do not need to be redistributed. It is similar when node is added. Therefore, consistent hashing largely reduces the load of moving data when nodes change.

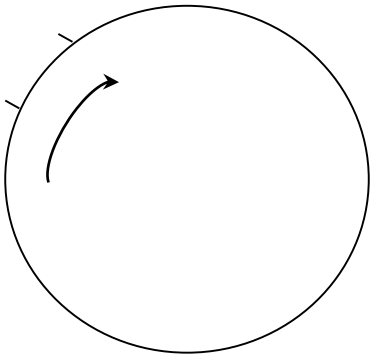


Figure1.1 Example a simple implementation of consistent hashing algorithm

And consistent hashing can be implemented by storing the points in a binary tree so that it supports the hashing to time with n nodes. Another implementation is to break the circle into equal-length intervals and improve the lookup to constant time.

2. P2P networks application

Consistent hashing is re-purposed [4] into address technical challenges that arise in peer-to-peer (P2P) networks. A key issue in P2P networks is how to keep track of where to look for a file, such as an mp3. This functionality is often called a "distributed hash table (DHT)." DHTs were a very hot topic of research in the early years of the 21st century.

First-generation P2P networks (like Napster) solved this problem by having a centralized server keep track of where everything is. Such a network has a single point of failure, and thus is also easy to shut down. Second-generation P2P networks (like Gnutella) used broadcasting protocols so that everyone could keep track of where everything is. This is an expensive solution that does not scale well with the number of nodes. Third-generation P2P networks, like Chord [4], use consistent hashing to keep track of what's where. The key challenge is to implement the successor operation even though nobody is keeping track of the full set of servers. The high-level idea in [4], which has been copied or refined in several subsequent P2P networks, is that each machine should be responsible for keeping track of a small number of

other machines in the network. An object search is then sent to the appropriate machine using a clever routing protocol. Consistent hashing remains in use in modern P2P networks, including for some features of the BitTorrent protocol.

3. Amazon Practice

Amazon implements its internal Dynamo system using consistent hashing [3]. The goal of this system is to store tons of stuff using commodity hardware while maintaining a very fast response time. As much data as possible is stored in main memory, and consistent hashing is used to keep track of what's where.

This idea is now widely copied in modern lightweight alternatives to traditional databases (the latter of which tend to reside on disk). Such alternatives generally support few operations (e.g., no secondary keys) and relax traditional consistency requirements in exchange for speed. As you can imagine, this is a big win for lots of modern Internet companies.

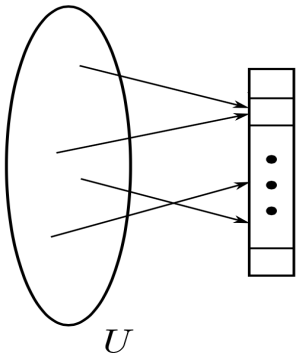
III. Traditional Hashing Algorithm

1. Hashing

Save items in a key-indexed table (index is a function of the key)

2. Hash function

h(K) = K mod M
K represents key, M means node. The keys are evenly mapped into M nodes (shown in Figure



3.2)

Figure3.2

3. Limitation

The traditional hashing algorithm doesn't work well when some node is added or deleted from the distributed file system. Such limitation increases the miss rate, leading a lot to system overhead. For example, if we set M = 7. We assume that Node 3 fails, the remaining nodes 4 to 7 have to be relocated in the system. In addition, the file originally mapping to node 5 has to change to node 4 (see the Figure 3.3).

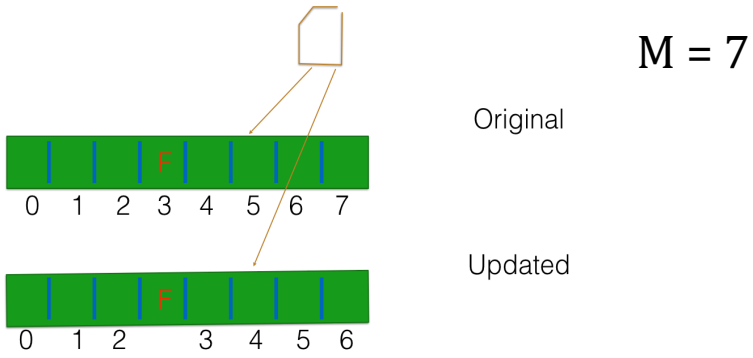


Figure3.3

IV. Methodology

This project tries to evaluate Consistent Hashing algorithm and traditional Hashing algorithm quantitatively in distributed file system. In order to do so, three key parts of work should be done. The first is implementation of Consistent Hashing algorithm and traditional Hashing algorithm. Secondly, a distributed file system simulator should be implemented. The last work is to evaluate these two algorithms in the simulator.

1. Algorithm implementation

Implementing these two algorithms is a relatively simpler part of work in this project. The important thing should be noted is that the two algorithms should use same hash function, since they should be compared on same basis.

1.1 MD5 hash function

We used MD5 hash function in both consistent hashing and traditional hashing algorithm to achieve comparable basis for our comparison. MD5 hash function exhibits some properties so that it is more suitable for our implementation.
First, there is very small possibility of getting two identical hashes of two different files, even if the files differ slightly. Second, MD5 hash function could hash the input in an unpredictable but deterministic way. That is, for a specific file, each time it will be hashed to a same value. Moreover, the length of the hash value is 128 bits in our implementation (or 160 bits in the most common implementation), and does not depend on the size of input.
Therefore, the results of hashing would be more balanced.

1.2 Virtual Nodes

In consistent hashing algorithm, we map all the files and data nodes on the edge of a same circle. And when data node is removed or added, only a small part of files need to reallocate. However, the objects are basically randomly mapped on the circle. As a result, it is possible that some nodes may contain more files whereas some contains less.

Virtual nodes are introduced here to make the allocation of files more balanced. Instead of mapping one data node on the circle, N (number of replicas of data node) points are mapped on the circle representing the original data node. Therefore, the intervals containing the files, which are allocated to a specific data node, are more spread out on the circle, and the allocating results would become more balanced to all the data nodes. And the balance would be improved as the number of virtual nodes increases.

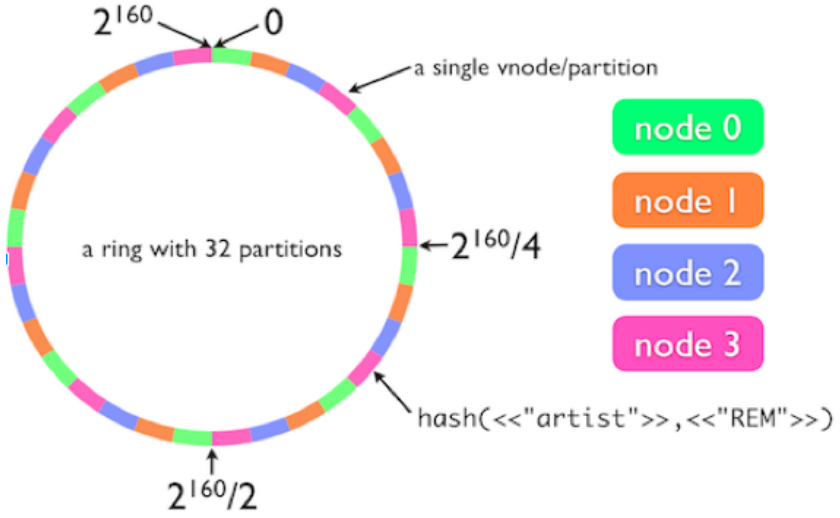


Figure 4.1 Virtual Nodes in Consistent Hashing

2. Distributed file system simulating

2.1 The mechanism of hash in distributed file system

As traditional file system, a key issue for distributed file system is how to find the location of a file in distributed environment. Obviously, it is much more complex in distributed file system, since the file might be stored in different disks of different storage nodes. That means distributed file system should have an effective mechanism to map a file into the location, where it is stored, according to filename, file path and offset. One of the main methods is Hash algorithm. That means if a client wants to access a file, it can calculate the positon of a the file it want to visit.

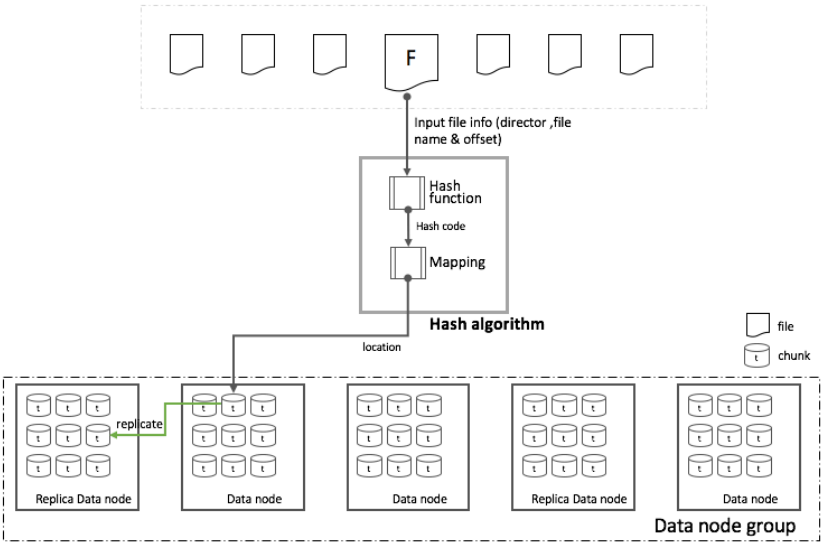


Figure 4.2 Mechanism of hash algorithm in distributed file system. In distributed file system, all files are stored in hundreds, even thousands of different servers, which are called data node and connected by Ethernet networks. Generally, a data node has several disks to store data and, for convenience, this storage space is divided into mass small blocks, called chunk. Actually, files are stored in this large amount of chunks. Hash algorithm help the file system to map a file into the location of chunk where it is stored. It should be noticed that a file might be split into multiple parts to be stored in serval chunks, if the file is larger than the size of a chunk.

2.2 Simulation

To set up a distributed file system, a large amount of resources are needed, such as many servers and Ethernet networks. To find and manage these massive resources might make this project unpractical and too complex to develop.

Fortunately, as the description of 2.1, to compare Hash algorithm, we only focus on how to map file into different location effectively. So the impact of networks and reading and writing physic disks become insignificant, even should be reduced or eliminated. Therefore, we can simulate the three key components—data node, trunk and file.

This project plans to implement a software simulator to simulate the whole mapping process, which running in a single PC. That means Ethernet networks and huge physic servers are not needed. It simplifies the model greatly. That can help us to eliminate the impact of other insignificant factors, such as unstable Ethernet and disks. And, more importantly, help us to focus on the two hash algorithms themselves. Also, this simulator should simulate three key operations—mapping files into different locations, writing and reading data in disks, duplication mechanism. Finally, a much important situation, data note crashes and joins frequently, should be simulated too, since it impacts the works of Hash algorithm greatly.

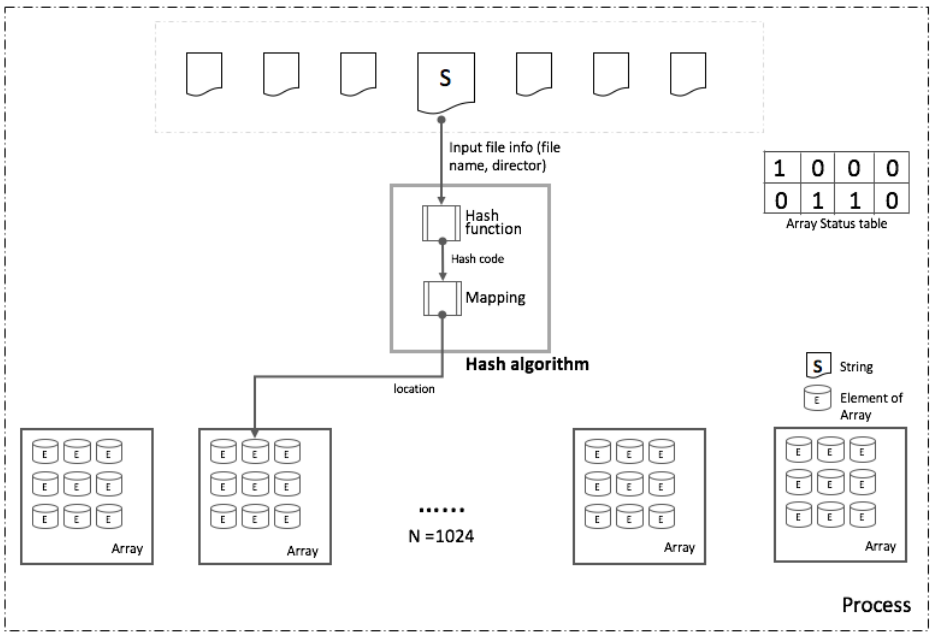


Figure 4.3, Simulation to distributed file system for Hash algorithm.

- Key components simulation

As the figure 4.2 shows, this project uses array, elements of that array and string to simulate, respectively, data node, trunks and file. Each string will be set to a combination of filename and directory.

- Duplication mechanism

In order to simulator the full recovering process of node crashing, the duplication mechanism should be also implemented in this project. This project stores two duplication and holds strict consistency. When a node crashing, all the data stored in that node will be recovered in left nodes.

- Operations simulation

First, for mapping operation, this project directly calculates the location, means which entry of which array, of a string according to the value of that string.

Secondly, this project uses adding or getting an element to or from an array to simulate the writing or reading file operations. Note that, this project will add some nonsense instructions to slow down the adding and getting operation, since writing and reading files from disks is much slower comparing to adding and getting an element.

- Data node joining and crashing

Large amount of array is used to simulate a lot of data nodes, so we can disable or enable an array to simulate crashing or joining of a data node. An Array Status Table is set up for this job. Each element of this table represents an array. If the value of an element is 0, the array cannot be used. Oppositely, if the value is 1, the array works. Therefore, this project can randomly

frequently change the value of an element to simulate the joining or crashing event of a data node.

3. Evaluation

3.1 Performance

Two properties are much important in distributed file system environment, since the hash algorithm in this environment is used to hash effectively distribute the data across the data nodes. Therefore, this project will compare quantitatively consistent hash and traditional hash algorithm from these two aspects.

- **Balance**

"Balance" means all data in a file system should be stored in different data nodes as evenly as possible. That can help the distributed file system takes full advantage of resources of all data nodes, such as networks, CPU, storage capacity.

- **Smoothness**

"Smoothness" means the amount of data, which should be moved to a new location after data node changes, should be as small as possible. That can help the distributed file system run more stable when the data node changes. When a data node is added to or removed from the group of data nodes, the range of the hash algorithm would change. In this situation, some of data will be hashed to a new location. That would be a disaster, if these data were not transferred to the new location, because the client of this file system will not get the accurate data according to the file information. So a distributed file system has to fix this issue by transferring all data to correct locations as needed.

3.2 Availability

As section 3.1 indicates, adding or removing a data node has a significant impact on distributed file system. Before moving data to correct location, system cannot work normally, so you can consider the time spent on moving data as system recovery time. This project will calculate the system recovery time respectively and compare the availability of the system in these two hash algorithms situation.

V. Conclusion

1. Impact of Number of Virtual Nodes On Consistent Hashing

1.1 Impact on Balance

Given certain number of nodes and number of disks, we select different number of virtual nodes to observe the impact of virtual nodes on the balance of all the files in our simulated file system. We measure the balance as the standard deviation of number of files allocated in different data nodes. As shown in the picture below, the balance index decreases dramatically when number of virtual nodes increases from 1 to 50. After the number of virtual nodes increases to 1000, the standard deviation gradually approaches to 0.

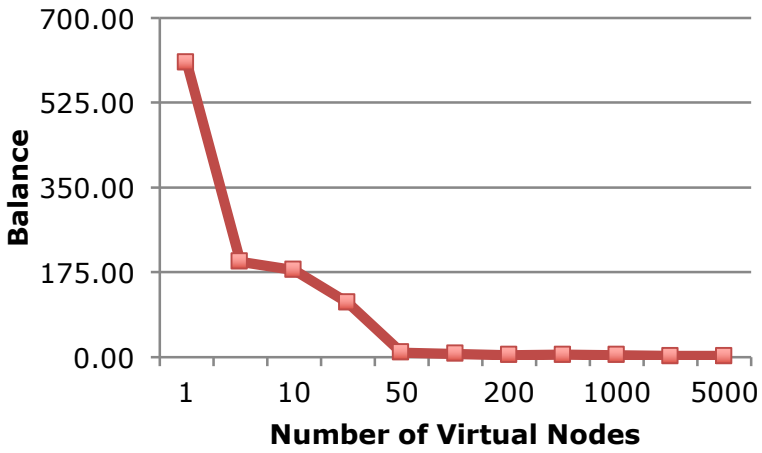


Figure 5.1 Impact of number of virtual nodes on balance

1.2 Impact on Time of Adding Node

More virtual nodes in implementing the consistent hashing algorithm would improve the balance of allocating files to data nodes. However, it is not necessarily better to have more virtual nodes. While adding a new node, the system would not be available in order to reallocate the files already in the file system.

In fact, we simulate the event of adding a new data node to system and calculate the recover time of relocating the existing files which map to a different data node. As shown in Figure 5.2, the time for adding a new node increases significantly when the number of virtual nodes is above 1000. And the benefit brought by increasing the number of virtual nodes is trivial when it reaches 1000, so that we select 1000 as the number of virtual nodes in our simulation.

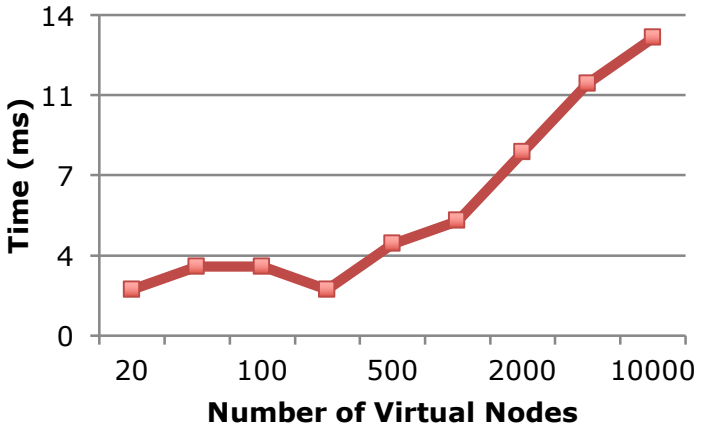


Figure 5.2 Impact on Time of Adding Node

2. Comparison of Performance

2.1 Balance

First, we compare the balance between traditional hashing and consistent hashing algorithm based on the standard deviation of number of files allocated on different nodes. Since we use

MD5 as our basic hash function, for traditional hashing algorithm, it is supposed to randomly get a distinct hash value for each file. Therefore, it would exhibit well-balanced property, especially for large number of data nodes. For consistent hashing algorithm, as number of data nodes increases, the balance is also improved. In general, as data nodes reach to 512, the result is well balanced and any increase of number of nodes would not improve the result significantly. Overall, there is no obvious difference between these two algorithms in terms of balance.

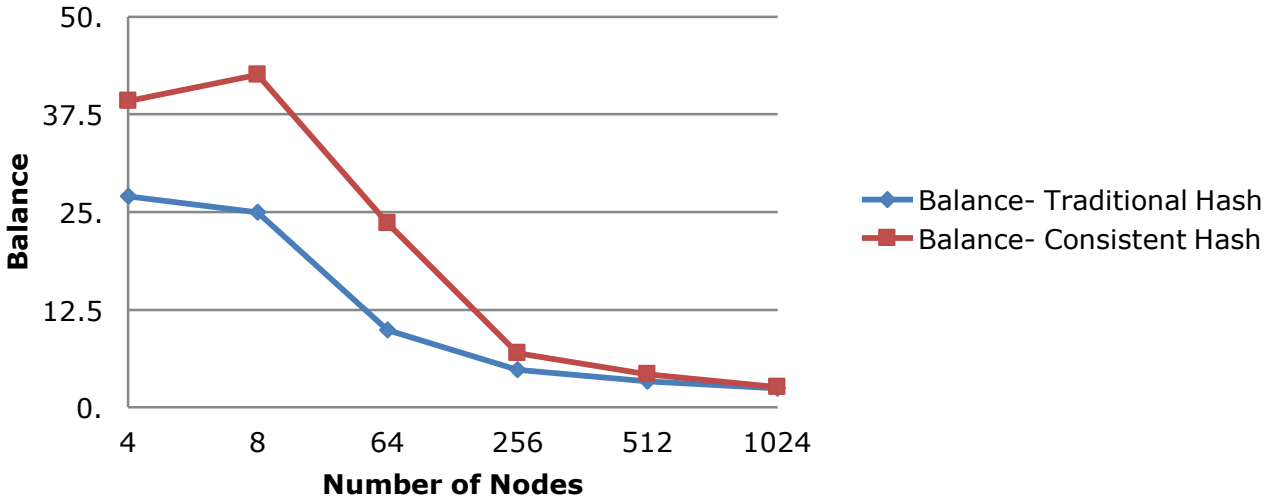


Figure 5.3 Balance of Traditional and Consistent Hashing

2.2 Smoothness

Apparently, a big problem of traditional hashing algorithm is that a huge amount of files need to relocate when data nodes changed in the file system. We first simulate the change of adding a new data node to system and calculate the number of chunks that would be hashed to another data node. As shown in Figure 5.3, when number of data nodes increase to 512, almost all the chunks in the file system need to relocate for traditional hashing algorithm. This would be a disaster for causing huge overhead when a new data node is added. On the contrary, only a small part of chunks need to relocate for consistent hashing algorithm and that part of chunks becomes trivial when number of data nodes increases to 512 because the intervals on the circle corresponding to that part of files becomes smaller.

The situation is similar for a node crashes or is deleted in the system as shown in Figure 5.4.

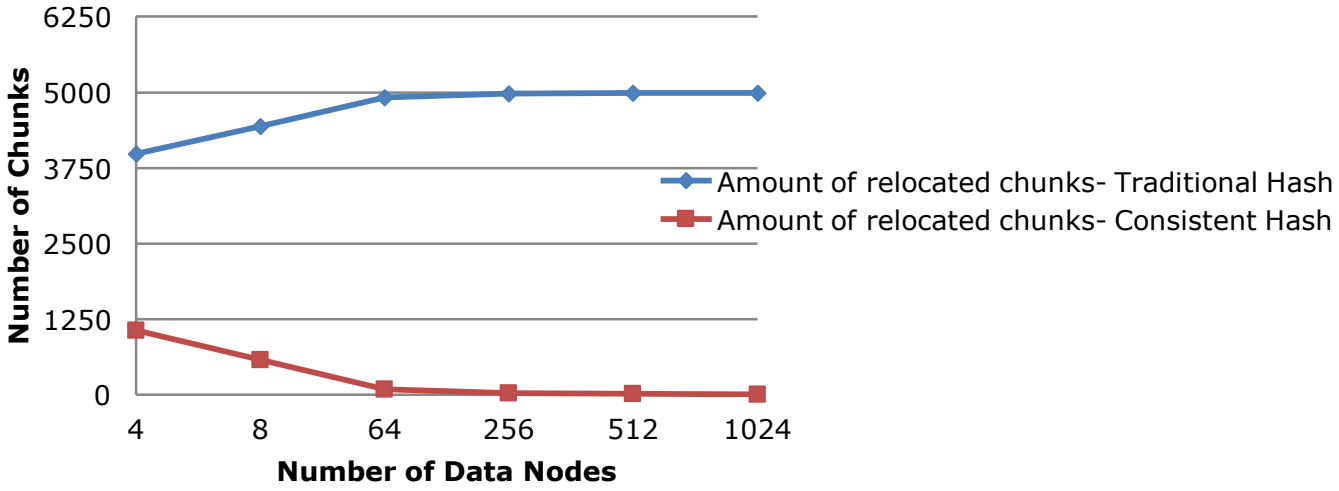


Figure 5.3 Number of Chunks Relocated When a New Node Added

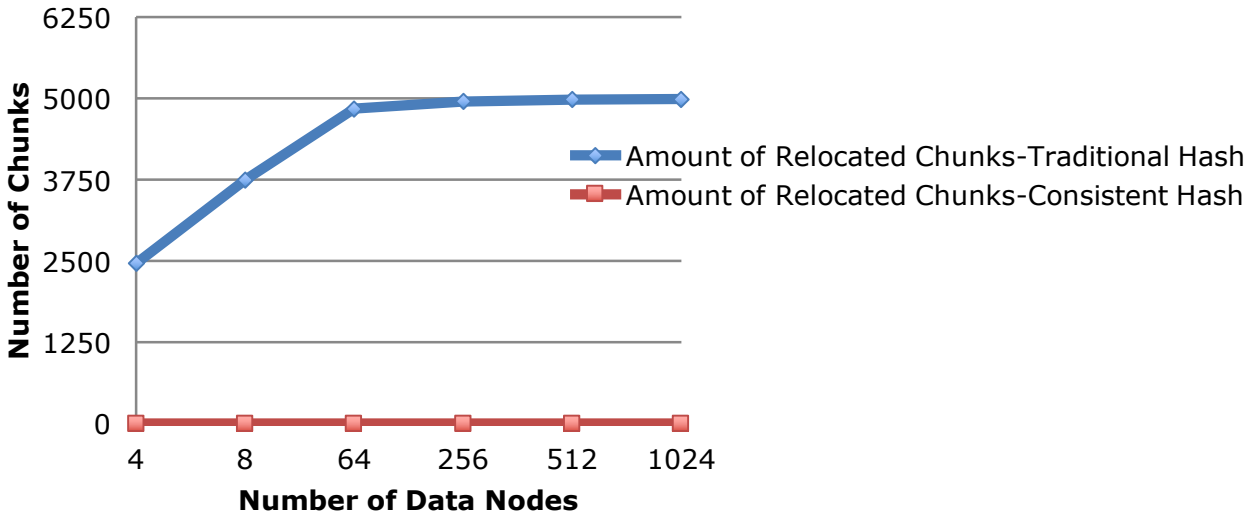


Figure 5.3 Number of Chunks Relocated When a Node Crashes

3. Availability

When data nodes changes, the system typically needs time to relocate the files that would be hashed to a different data node and then the system would be unavailable for some time. We simulate the time needed to relocate the files to compare the availability of file systems using these two hashing algorithms. The results are similar to that of smoothness. For traditional hashing, recover time increases significantly until the number of data nodes reaches to some point that almost all the files need to relocate, and the availability cannot satisfy the user when data nodes changes a lot. However, the recover time decreases as the number of data nodes increases.

It is worth noting that we focus on the trend of these two figures because the actual recover time depends on many aspects such as system configuration. We can only simulate the situation and compare the result we get on same basis. The trend itself is illustrative, whereas concentrating in the actual number is pointless to some extent.

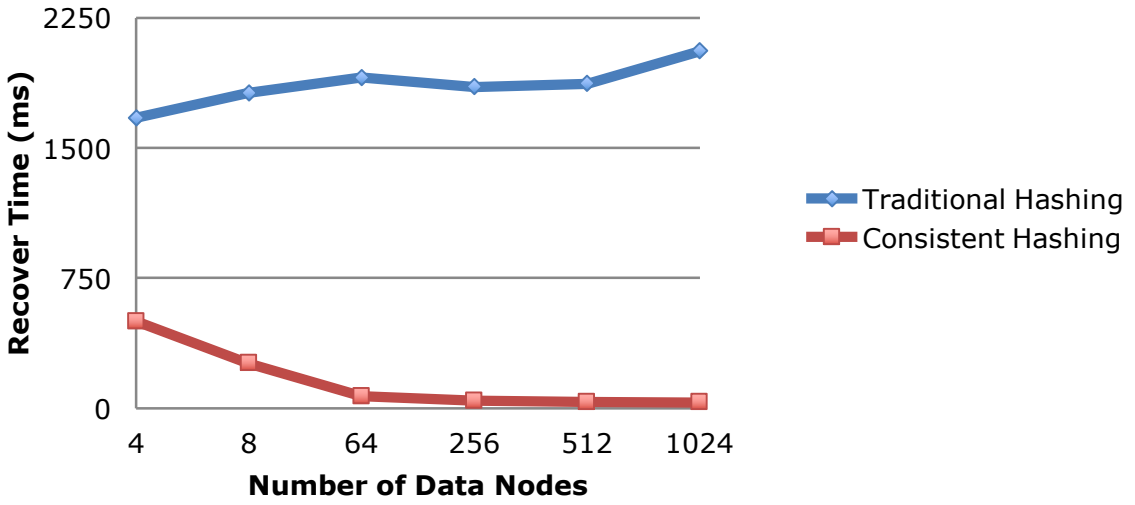


Figure 5.4 Recover Time When a New Node added

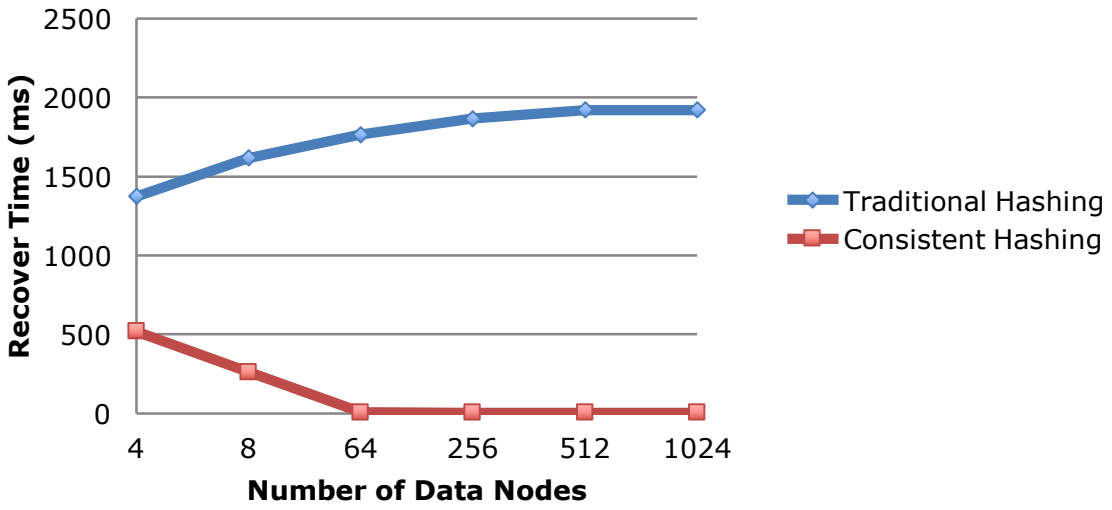


Figure 5.4 Recover Time When a Node Crashes

In conclusion, based on the result of our simulation, there is no significant difference in balance for these two algorithms. Consistent hashing would be well balanced when number of virtual nodes reaches 1000. More importantly, file system using consistent hashing algorithm would be better in terms of smoothness and availability, especially for large number of data nodes. In this case, we could conclude that consistent hashing algorithm is more suitable for distributed file systems.

Reference

[1] Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent Hashing and Random Trees." *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing - STOC '97* (1997).

[2] Karger, David, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. "Web Caching with Consistent Hashing." *Computer Networks* 31.11-16 (1999): 1203-213.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. “Dynamo: Amazon’s Highly Available Key-value Store”. Amazon.com

[4] Ion Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. “Chord: A scalable peer-to-peer lookup protocol for internet applications.” IEEE/ACM Transactions on Networking, 11(1):17–32, 2003.