# Implementation of a Distributed File-System with Fault Tolerance

Shailesh Samudrala, Shanmuganathan Navaneetha Krishnan

*Department of Electrical and Computer Engineering, University of Florida*
*Gainesville, Florida, USA*
shailesh2088@ufl.edu
s.navaneethak@ufl.edu

**Abstract:**

**This paper implements a single level Distributed File system on the basis of client-server architecture using Distributed Hash Tables. The servers are implemented as a peer-to-peer network where all servers are equals. The client connects to any one of these nodes, and the data stored by the client is distributed across the network. This implementation is based on File-system in User Space (FUSE). FUSE is a loadable kernel module which lets non-privileged users implement their own file systems in user space without editing kernel code. Our implementation addresses the following concerns: Load Balancing, Decentralization, and Availability.**

## I. INTRODUCTION

In typical client-server architecture, all clients connect to a central server which controls all communication between clients. However, a client-server system is not very scalable because the load on the server increases with the inclusion of every new client. Also, if the server fails, then the whole system fails, and client cannot communicate with each other.

However, a peer-to-peer system consists of many nodes, or peers, which have an equal role in the system. Each peer acts as both a client and a server and communicates directly with each other. Thus there is no reliance on a central server. This makes peer-to-peer systems easily scalable and fault-tolerant. Peers can enter and leave the system without much effect on the overall system performance. Peer-to-peer systems are widely used. Napster [6], Gnutella [7], and LimeWire [10] are a few examples of applications based on peer-to-peer systems.

Peer-to-peer systems can be of two types: unstructured and structured. In an unstructured peer-to-peer system, there is no algorithm which controls the organization of the system. However, structured peer-to-peer systems employ a consistent algorithm that controls the organization, routing and optimization of the system. The most common type of structured peer-to-peer system is a Distributed Hash Table (DHT).[9]

Distributed Hash Tables are a class of decentralized distributed systems that provide a lookup service similar to a hash table which stores (key,value) pairs. Any participating node can retrieve the value associated with a key. This paper implements the Distributed File System as a Distributed Hash Table. Distributed Hash Tables are widely used to store large amounts of data. A few examples are: Facebook's Cassandra, and Amazon's Dynamo.

*Overview or our implementation:*

In our implementation we use Distributed Hash Tables to store the data of the file system. The File system is implemented using Filesystem in User Space (FUSE) which is a loadable kernel module which lets non-privileged users implement their own file systems in user space without editing kernel code. The highlights of our implementation are:

- *Load Balancing*: The data stored by the clients is evenly distributed across the servers so that a single server does not get overloaded. This is achieved using a hash function which distributes keys evenly over the nodes in the system.
- *Decentralization:* There is no central server which dictates how peers communicate with each other. All nodes are equal, and communicate with each other either directly, or through neighboring peers.

- *Availability*: We achieve a degree of fault tolerance by replicating the data of one node in another node. If any particular node fails, or disconnects from the network, then requests to the data that was stored in that node can still be served by the node in which the replicated data was stored.
- *Routing:* Each node maintains a table of its neighbors as well as sample nodes from across the network in an effort to minimize the number of hops needed in order to communicate with another node. This significantly traffic in the network, as well as improves the latency of fetching data from another node.

*Potential Issues:*

The potential issues regarding our implementation are the following:

- It does not consider the possibility of new nodes entering the system.
- The system does not address the possibility of a node and its successor failing at the same time
- It also does not provide any solution to the fate of the client when the node to which the client is connected fails.

## II. RELATED WORK

Since Distributed Hash Tables are popularly used, a lot of research has gone into optimizing the system, making it more efficient, fault tolerant, and easily scalable. Ion Stoica et.al proposed a distributed lookup protocol, called Chord.[1]

Chord is a protocol designed for efficient location of a desired data item in peer-to-peer systems. Chord uses consistent hashing to assign keys to nodes. Consistent hashing ensures a high probability of keys being evenly distributed across the servers. To improve scalability of the system, Chord requires that a node needs to know only a small amount of routing information about other nodes. In an N-node network, each node maintains information about only O (log N) nodes in a table known as the Finger table, and a lookup requires O (log N) messages.

Chord protocol also includes a Stabilization function. Each node runs this function periodically to learn about new nodes. Stabilization ensures that existing nodes are still reachable when new hosts join the network. Chord also maintains a list of successor nodes, of size r. This list contains that node's first r successors. If all r successors fail simultaneously, then the Chord system would be disrupted. However the probability of that happening is , where p is the probability that all r successor nodes fail simultaneously. The system can be made more robust by increasing the value of r, i.e. increases the number of successors in the successor list. The Stabilization function will correct the system in case of a node failure.

If a node voluntarily departs from the system, it transfers all its keys to its successor before departing. The departing node can also inform its successor and predecessor before departing, so that updates can be made, and the system can remain consistent. Suppose Node n has Node m as its predecessor and Node p as its successor. If Node N departs voluntarily, it informs Node m and Node p about its departure. Then Node p will update its predecessor to Node m, and Node m will update its successor to Node p, thus making the system consistent. Therefore, the Chord protocol provides features such as simplicity, correctness, robustness, scalability and good performance. The structure of a Chord-based distributed storage system is shown in the figure 1 below:

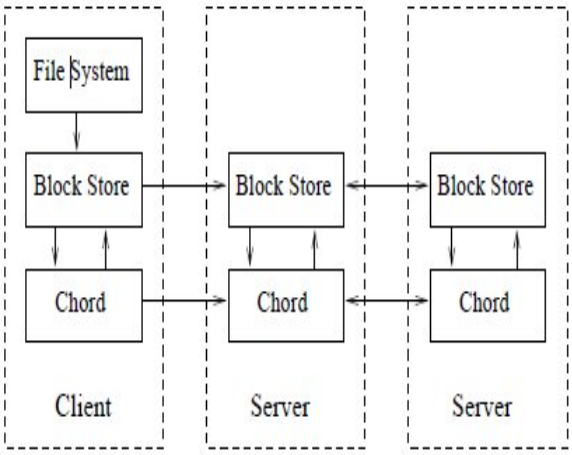


Figure 1: Structure of a Chord Based System. [1]

Other work on Distributed Storage systems includes the Freenet peer-to-peer storage system, which is also decentralized, symmetric and scalable.[4] The Ohaha system is another example of a peer-to-peer storage system.[5] Archival Intermemory protocol uses an off-line computed tree to map logical addresses to machines that store the data. Protocols for distributed lookup similar to Chord also exist. The distributed data location protocol developed by Plaxton et.al is very similar

to the Chord protocol. The Tapestry protocol is also a Chord-like protocol.[8] Pastry, which is a prefix-based lookup protocol, is another example.[2] Napster and Gnutella have also developed distributed lookup protocols to find data in a distributed set of peers. These protocols base their lookup on user-supplied keywords. This presents difficulties in both systems. In the case of Napster, a central index is used, thus resulting in a single point of failure. If the central index fails, then the whole system fails. In Gnutella's case, a node floods each query over the entire system. This increases the communication and processing costs of the system, which increases drastically when the system gets bigger.

We have used the Chord protocol as a reference in our implementation. We ensure efficient routing of requests similar to Chord. We have also implemented a successor list, as it is implemented in chord, with r=2 i.e. 2 successors. We have also achieved a high degree of decentralization and fault tolerance. However, our implementation does not provide for scalability of the system, and remains an area to be worked on.

## III. IMPLEMENTATION METHODOLOGY

As stated earlier, we have implemented our Distributed Storage system in the form of Distributed Hash Tables which store (key, value) pairs using the Chord protocol as a reference to develop our own algorithm to handle load balancing, fault tolerance, decentralization and routing. In our implementation, we have implemented server nodes arranged as a structured peer-to-peer network. We have also implemented the client module which will be the gateway between the user and the Distributed File system. The architecture of our Distributed File system is shown in Figure 2.

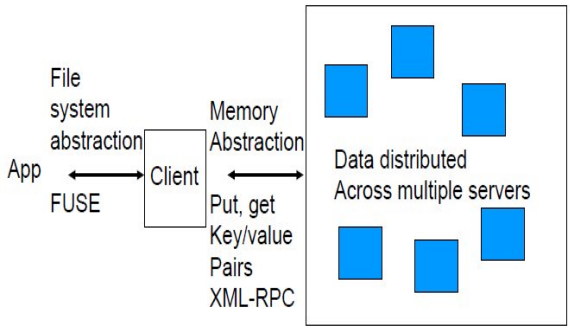


*Figure 2-Distributed File system Architecture*

The client module is implemented using Filesystem in User Space (FUSE). The FUSE module itself is embedded within the kernel. The client module, however, is implemented in User Space. Any requests from the user to access data in our file system are redirected by the kernel to the client module running in User Space. This module handles all interactions between the User and the Distributed File system. All major file system operations are supported by the client module. However, the client module is able to implement only a single level of directories. The client module connects to any one of the nodes in the peer-to-peer network. The client module interacts with this node alone. The node, upon receiving data from the client module, will determine the node where the data must be stored, and redirect the data to that node. All this is done transparently, so the user has no idea about where data is being stored or from where data is being retrieved. The client module interacts with the server using "put" and "get" operations. The interface between client and node is implemented using XML-RPC.

The client module organizes all data into (key, value) pairs. A put operation stores a given key, value pair into the Distributed File System. A get operation retrieves the value associated with a given key from the Distributed File System. Every node has a unique 128 bit node id. When the client module needs to put a (key, value) pair into the file system, it uses a hashing algorithm to hash the key, thus obtaining a 128 bit object id. Using a good hashing algorithm is important to balance the load of the system. Our implementation uses the md5 hashing algorithm. Thus, after obtaining the object id, the client module issues a put statement to the node to which it is connected.

The structure of the put statement is as follows: put (object_id, key, value, time to live). Time to live determines the amount of time that piece of data will be stored within the File System. Also, the maximum amount of data associated with any given key is 1024 bytes. Thus the client module accordingly splits up large amounts of data into chunks of 1024 bytes before issuing a put command to the server. This ensures that data is distributed evenly across the system, as large files will now be stored in multiple nodes, instead of increasing the load on a single node.

The structure of a get statement is as follows: get (object_id, key). The client module issues this command to the server when data needs to be retrieved from the Distributed File System. Upon

receiving this request, the server node uses the object id to determine which node in the network is currently storing the required data, and then requests the data from that node, passing it along to the client. The client then re-orders the received data as required by the user.

The server nodes are implemented as Distributed Hash Tables, storing (key, value) pairs. All nodes are organized in a peer-to-peer network in a partial ring-like fashion. Our implementation uses a central controller to initialize the nodes in the network, and to provide each node with information about its successor and predecessor. Although this central controller can potentially be a single point of failure, the probability of this happening is extremely low, as the controller is used only in the nascent stage of the system, when the nodes are being setup for the first time. One the nodes are up and running, the central controller plays no further role, and nodes interact directly with each other in accordance to an algorithm that is known to every node in the system.

As stated earlier, the client interacts with a node using the "put" and "get" statements. When a client issues a put request to the server, the server receives this request, uses the object id to determine which node in the network should store the data, and redirects the data to that node. Similarly, when the server receives a get request from the client module, the server uses the object id to determine which node is currently storing the data, and requests the data from that node, thus transferring it to the client module. Thus, the two major operations performed by any node is storing and retrieving data.

Since the nodes are arranged in a peer-to-peer network, and there is no central server to route messages to and from a network, any node in the network should be able to route data efficiently to other nodes. To achieve this goal, each node maintains a table of node ids spread across the network, known as the finger table. It is important to mention that the node ids are assigned in an ascending order to the nodes in the network. Thus, a node's node id is always greater than its predecessor's node id and always lesser than its successor's node id expect when the node has the highest or the lowest node id in the whole network. The finger table is initialized to store node ids and address of 2 consecutive successors and 2 consecutive predecessors. It then samples the network to retrieve node id's and addresses of random nodes distributed across the network. This is done by the init_finger ( ) function. Thus, the finger table consists of the node ids and addresses of seven nodes from across the network.

The central controller initially provides every node in the network with the node ids and addresses of its successor and predecessor. Every node maintains a list of its two immediate successors and predecessors, called a Successor-Predecessor List (SPL). The list is populated using two functions get_successor( ), and get_predecessor ( ). When a given node invokes the get_successor ( ) function, it contacts its successor node and requests information about the successor's successor. The successor node replies with the node id and the address of its successor. Similarly, get_predecessor ( ) retrieves the node id and address of the node's predecessor's predecessor. The node ids of the two successors and two predecessors are stored in the Successor-Predecessor List (SPL). The node ids and addresses returned by the above two functions are also stored in the finger table. The finger table strives to minimize the traffic in the network by trying to reduce the number of hops required to contact other nodes.

In order to balance load across the network, data is distributed across all the nodes in the network. Thus, a node will have to send or receive data to or from other nodes in order to serve clients connected to it. In order to do so, the node must first locate the desired node, before communicating with that node. Finding the desired node is based upon the object id provided by the client. The File system is implemented such that data is stored in the node whose node id is larger and nearest to the object id of that data. Two possibilities arise here:

a) The required node is the node itself.
b) The required node is any other node in the network.

When a node is required to locate another node, it first compares the node ids of all the nodes stored in its finger table to the object id of the data to be stored or retrieved. Since the data is to be stored in the node whose node id is larger and nearest to the object id of that data, the node selects the node whose node id is nearest and smaller than the object id of the data from its finger table. It then checks if the node id selected from the finger table matches that of its immediate predecessor in the SPL. If it does, it means that the data should be stored by that node itself. Thus the first possibility is satisfied. If the selected node is not the predecessor, then the node issues a put statement to the node that was selected from the finger table. This will happen

recursively until the required node is found. The find_node ( ) function is called by the node to locate the desired node. The pseudo code of find_node ( ) is given below:

```
if object_id  <= node_id then
  if node_id = = object_id  then
    destination = current_node
  else if node_id < predecessor_node_id then
    destination = current_node
  else if  node_id > predecessor_node_id and
object_id  > predecessor then
    destination = current_node
  else
    find node from finger_table whose id is lesser
than object_id and return it as destination , if no id
in finger table is less than the data_key return the
highest id in finger table

if object_id > node_key then
  if object_id > predecessor and predecessor >
node_id then
    destination=current_node
  else if object_id <= successor then
    destination=successor
  else if successor < node_id then
    destination=successor
  else
    find node from finger_table whose node id is
lesser than object_id and return it as the destination
```

Now, we consider the possibility of node failures. Our implementation assumes that if a node fails, then the successor of the node doesn't fail at the same time. We cope with node failures by replicating the data stored in one node in its successor node as well. Thus, if a node receives a put request that is meant for itself, then it invokes a function put_backup ( ) to store the data in the successor node as well. Thus, if a node fails, then the data is still available in its successor node, which will be accessed by default, based on the design of the algorithm. Apart from data replication in successor nodes, every node runs a stabilize function periodically to detect node failures. If a node finds that its successor has failed, then it corrects the entries in the SPL and the finger table to point to its 2 new successors. The get_successor ( ) function is invoked here. The node whose successor has failed informs its predecessor to update its finger table and SPL by providing it with the updated values of its successor. Also, since the node has lost its backup data, it should transfer all its data into the new successor. This is done buy invoking the put_backup ( ) function. Every node also checks if its predecessor has failed. If the predecessor has failed, then the node updates its finger table and SPL to reflect the 2 new predecessors The get_predecessor ( ) function is invoked here. The node also informs its successor to update its SPL and finger table by providing the new values to the successor. If the node finds that a node other than those in the SPL in the finger table does not respond, it just deletes the value from the finger table, and includes the node id and address of another node from the network. If the predecessor of a node fails, then it means that the data of the failed node is now stored only in the successor node. If the successor node also fails, then that data cannot be retrieved and the system will not display correct behavior. Thus, when a node finds that its predecessor has failed, it will transfer all of its data into its successor, thus creating a backup of the failed node's data. There is also a possibility that a node fails in between the time the nodes run the stabilize function. To address this case, every put and get operation first checks the availability of the node before issuing a request to that node. The pseudo code for stabilize function is given below:

```
check successor:
  if failed then
    Make second successor as first successor and
make the successor of the new successor the second
successor. Also for backup copy all the data in the
node to the new successor. Inform predecessor to
update second successor.

check predecessor:
  if failed then
    Make the second predecessor as predecessor and
make the predecessor of the new predecessor the
second predecessor. Also for backup copy all the
data in the node to the new successor. Also for
backup copy all the data in the node to its successor.
Inform successor to update its second predecessor.

check other nodes in finger table:
  if failed then
    update the finger table using find_succ.
```

## IV. TESTING AND RESULTS

The system was run with fifty nodes, which were initialized by a central controller. Each node was run as a different process. To test whether the system was working correctly, we connected the client to a particular node, and performed typical file system operations such as making directories, storing and editing files, changing ownership permissions etc. We concluded that the system was running correctly and efficiently. We then tested the fault tolerance of

the system by removing nodes randomly from the system. We then requested data from the failed nodes before the stabilization function was invoked. This request returned an invalid address exception, because put and get operations first check the availability of the node. The same operation was again performed after stabilization of the system, and the requested data was retrieved. Therefore, the system was found to run correctly, hence verifying the system's fault tolerance.

However, due to shortage of time, our testing has not been very comprehensive. Ideally, testing should have included analysis of the path lengths for communication between nodes across the network, time taken to access data requested by the client, and testing the system with multiple clients connected to the Distributed File System.

## V. EVALUATION

The Distributed File system which we have implemented efficiently distributes data across the nodes in the network, thus achieving a desirable degree of load balancing. Load balancing is also ensured by dividing large pieces of data into chunks of 1024 bytes. The system is decentralized, with the central controller being used only to initialize the nodes. The system is characterized by a good degree of fault tolerance, resulting in availability of data even when nodes fail or disappear from the network. The stabilize function works efficiently to detect and rectify the system in the event of node failures. The use of the finger table reduces network traffic and results in faster access times to the desired data items.

The limitations of our system are:

- The system fails if a node and its successor fail at the same time.
- We have assumed that a fixed number of nodes initially exist in the system. We do not address the possibility of new nodes entering the system. Thus, our implementation does not address the scalability of the system.
- Another case which is not addressed is the fate of the client when the node to which the client is connected fails.
- The file system implements only a single level of directories.

Our implementation requires more intensive testing and analysis; it has the potential to develop into an efficient and usable Distributed File System algorithm.

## VI. CONCLUSION AND FUTURE WORK

We have successfully implemented a Distributed File System which successfully addresses concerns such as load balancing, decentralization and fault tolerance. Load Balancing is achieved by using hashing functions to distribute data evenly throughout the network, as well as breaking up large pieces of data into smaller chunks. Decentralization is achieved by organizing the Distributed Hash Tables in a structured peer-to-peer network. An appreciable degree of fault tolerance is also achieved by replicating data of one node in its successor node, and by stabilizing the system at regular time intervals. However, our system has a few shortfalls. The system is not scalable, and it assumes only a fixed number of nodes are initialized. It does not consider the possibility of new nodes entering the system. The system does not address the possibility of a node and its successor failing at the same time. It also does not provide any solution to the fate of the client when the node to which the client is connected fails. Also, our system has not been tested extensively. Additional testing and results analysis is required.

Future work on our implementation will involve addressing the limitations mentioned in the previous section. Scalability of the system is a very important area which we need to work upon, to make our implementation more realistic. The file system could be extended to support multiple directory levels. Caching of recently fetched data in the client is also an area that needs to be researched upon. Caching enables faster delivery of data that is frequently accessed, reducing the time the user has to wait for the data to be delivered by the system. Our implementation could also be extended to support concurrent operations in order to support multiple clients. This will require the system to support atomic operations on objects stored in the file system and support for locking files. Security of the file system is another important area of future research.

## VII. ACKNOWLEDGEMENTS

We would like to thank Professor Renato J. Figueiredo for providing us with this opportunity to work on such an exciting and interesting area. We would also like to thank Mr. David Isaac Wolinsky for his help and guidance which helped us complete this project. Finally, we would like to thank Srihari,

whose valuable advice helped us find solutions to difficult situations.

## VIII. REFERENCES

[1] Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications - Ion Stoicay , Robert Morrisz, David Liben-Nowellz, David R. Kargerz, M. Frans Kaashoekz, Frank Dabekz, Hari Balakrishnanz

[2] Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems - Antony Rowstron1 and Peter Druschel

[3] Peer to peer systems: http://en.wikipedia.org/wiki/Peer-to-peer

[4] Freenet: http://en.wikipedia.org/wiki/Freenet

[5] Ohaha: http://sourceforge.net/projects/ohaha/

[6] Napster: http://en.wikipedia.org/wiki/Napster

[7] Gnutella: http://en.wikipedia.org/wiki/Gnutella

[8] Tapsetry (Distributed Hash Tables): http://en.wikipedia.org/wiki/Tapestry_(DHT)

[9] Distributed Hash Tables (DHT): http://en.wikipedia.org/wiki/Distributed_hash_table

[10] LimeWire: http://en.wikipedia.org/wiki/Limewire