**Design Document**

# Introduction

This Software is an implementation of a Distributed Hash Table system and it mainly makes use of a Hashing Function to calculate a random value within a range. Separate Copies of data value pairs will be stored or distributed across Peers or Clients. The Value obtained from this Hashing Function will be used to determine the Peer where the Data will be present and connection is made to that Peer to get the value of the data stored .This Application mainly does 3 operations : storing a key/value pair, get value from a key and delete a key value entry.

# HashTable

In computing, a hash table (hash map) is a data Structure used to implement an Associative Array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

The Logic for Hash Function in this Application is contained in a method **hashingFnc()** and in a class **HashingClass.java**.

In a well-dimensioned hash table, the average cost for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

# Distributed Hash Table

A distributed hash table (DHT) is a class of a decentralized distributed systems that provides a lookup service similar to a hash table pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

The Implementation for get, put, delete operations are contained in **ClientListenThread** Class.

# Logical Organization Of the System

This system mainly consists of 2 components : (1) **ClientListenThread** (get,put,delete operations are done here)(2) **HashingClass** (Hashing Implementation done here)

(1) **ClientListenThread**:

The Get,Put,Delete operations on the Key/Value Data pair would be done here. The Data is stored in a global Data Structure called dataMap . The **get** operation is implemented in **getDataFromMap**(),put operation is implemented in **putDataintoMap**() and delete operation is implemented in **deleteDataFromMap**()

(2) **HashingClass**:

The Logic for Hashing is present in **hashingFnc**() inside **HashingClass**.java.The Logic For Hashing used here is :

*hashValue = hashValue + ((R\*hashValue) + entryToHash.charAt(i));*

where R is a random prime Number (31) and multiply it .The Prime Number makes sure that a random number is generated.

# Known Risks/Tradeoffs

1. Issue of Scalability when there are a large number of Threads

Since we are creating a new thread every time a new connection is created which is a resource intensive thing to do , when dealing with huge number of requests as is the case when dealing with large scale systems , this approach can present a problem.This approach of using Threads for every connection was used because it is easier to maintain and debug for relatively normal loads.

1. Lack of Persistence means that all the details that are saved in the DataMap in Peers would be lost if the Client is shut down for any reason.

# Features/Characteristics

1. Support for handling multiple requests from multiple clients concurrently.
2. Can Support User Defined Number of Peers/Clients.

# Possible Improvements/Extensions

1. Utilizing Non-Blocking mode of communication instead of Blocking and thereby taking away the need for creating an individual Thread for every socket connection.
2. Introducing Persistence so that the data is not lost even if the Index Server is restarted.
3. Making use of Thread Pools instead of creating a new Thread every time a connection is made. Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead. An important advantage of the fixed thread pool is that applications using it *degrade gracefully.*