# CS550 Advanced Operating Systems
# Programming Assignment 2
# Design

submitted by:
Chiranjeevi Ankamredy
A20359837

The assignment is on implementing Distributed Hash Table (DHT) and performing put, get and delete operations on the hash table and Multiple clients accessing the DHT..

We Assume that servers and network is static. There are 8 Clients, each client will act as client and server,each server has its own internal Hash table which stores Keys and values of Strings. Each of the 8 peers does the following operations on Serevr:

1. put(key, value)
2. retrieve(key)
3. delete(key)

Each peer hashes the key and then invokes a particular operation on the server that represents the hashed value.

To provide handling of concurrent requests, servers are implemented in multi-threading in java.

**The above scenario is implemented in JAVA using the concept of sockets, Multithreading for communication.**

**Functionalities:**

**1. PeerServer**

Here, we have a config file containing IP Address and Port of 8 peers since each peer acting as a client and server. When a server is initialized, it contains a server port which is given as a Static The Server receives a message from peers to do either put, get or delete operations on its hash table and does the appropriate operation on server. It is multi-threaded to take up multiple peers requests and perform all the three operations. The Server also ensures resilience of key-value pairs by replicating the key-value pair on the neighbor node.

**2. PeerClient**

The client also reads the config file and connects to all the servers during initialization, and maintains the connections until the user chooses to exit. Here, user will give Hash key and connetct to the Server.This reduces the overhead of having to connect every time the client invokes an operation on one of the 8 servers. The peers acting as clients do the following operations

**a.Hash(key):** The hash function returns a randomized hash value for any key. This (hash_value)%8 gives a value 0 to 7 , each value representing one of the 8 servers given in Config File. We refer to this value as node.

1. **put(key, value):** The Client doing put(key, value) hashes the key and sends a message that invokes the 'put(key, value)' operation on the server that represents the node. It returns 'Sucess' on successful PUT operation on the Server and 'Failed' if the 'put' is failed.

2. **get(key):** The Client doing get(key) hashes the key and sends a message that invokes the 'get(key)' operation on the server that represents the node. If the key exits at the server, then the client receives the value, else it receives the message "Invalid Key".

3. **delete(key):** The Client doing delete(key) hashes the key and sends a message that invokes the 'remove(key)' operation on the server that represents the node. It returns 'Deleted' on successful DELETE operation and 'Invalid key' if the 'delete' is failed.

**MESSAGE_FORMAT:**

Client and server interact through messages for Each operatiin. To invoke an operation client sends a message to server. The message is a string containing information about what operation to invoke at the server. The message that Client sends to the server is a string of the format " KEY {VALUE}". The message contains value if it is a PUT operation.

The message size varies with the size of key. VALUE is a string of 1000 bytes and is generated randomly by using commons-lang3-3.4.jar package.

OPERATION_ID can be 1, 2 or 3.

**1** for PUT (key, value) : "c 1 KEY VALUE" (if it is to invoke put (KEY, VALUE) operation)

**2** for GET (key) : "c 2 KEY" (when invoking get (KEY) operation)

**3** for DELETE (key) : "c 3 KEY" (when remove (KEY) is invoked)

The Server receives the message and interprets it and performs the operation based on OPERATION_ID.

After receiving the message, if the server interprets the message as PUT operation, it replaces the "OP" in the message and forwards the message to its neighbor node for replicating the key-value pair, so the neighboring node knows it is from the server and it stores the replica of key-value pair in its hash table.

**Multi-Threading and Concurrency:** Servers are expected to handle operations request from multiple clients at a time. So, to ensure concurrency on the server side multi-threading is used. The server creates a new thread for each client connection. If server fails to do Hash key Operations then it also creates an extra thread by executing Resilience.java to replicate the key-value pair at neighbor node.

The communication is synchronized between the clients and servers. The client invokes a certain operation and waits for the server to respond to the message.

**Improvements and Extensions to the program:**

Multi-threading at peer server is implemented by generating a new thread for a client connection, since there are only 8 peers to considered, thread generation doesn't seem to add that much

overhead, but when there are large number of peers communicating and Thread pooling is often more convincing approach to handle multi-threading, to reduce the significant overhead of thread creation for each client connection, Here, we haven't create locks on process because since we have only 8 servers, that can be done by creating a pool of threads using **new CachedThreadPool** method for creating threads and also Event driven programming can give a better performance at a larger scale of peers.

To ensure further data resilience, the replication can be done across several nodes rather than a single next node.