Design Document

Communication between Server and Client:

The communication between the peers and the server happens using Java Socket API. We use the technique of **ObjectInputStream** and **ObjectOutputStream** of the Socket API for communication amongst the nodes. Both the techniques are capable of transferring information using java objects only.

We'll go through the design of the system by referring each module.

1. **Network Configuration file**
   - The network configuration file provides the information of the peers in the network. It provides the program with the IP address of the peers which are there in the Distributed Hash Table network. The name of the network configuration file must be "network.config".
   - The key NODES contains comma separated IP address of the peers in the network.

2. **Hash function**
   - The hash function takes as input Key which is a string of max. 12 characters and returns an integer value which denotes the ID of the node where the Key can be stored or it can be accessed for retrieval/deletion.
   - So, the steps to find the node ID from the input key is –
     a) Perform hashCode of that Key. This is done by using Java's hashCode() method of Object class. The hashCode is a large integer value and it is always same for a given string. Two strings will not have same hashCode.
     b) After we get the integer hashCode, we divide it by total number of nodes present in the network (Total no. nodes can be retrieved from configuration file). In short, we perform MOD of hashCode and nodeCount.
     c) The result of this MOD operation is returned by this hash function to the calling function.

3. **Request**
   - Request class is used whenever a peer client wants to place a request (For example, PUT, GET, DELETE, etc.) to another peer server in the network. This class has two fields - **requestType** and **requestData**.
   - Whenever a peer client wants to place a request, it creates a Request object, sets it **requestType** which is basically a command like PUT, GET, DELETE, etc. and sets the **requestData** to appropriate data it wants to send in the request (For example, Key and Value in PUT request) and sends the object using **ObjectOutputStream**.
   - A peer server tracks incoming Request object using **ObjectInputStream**.
   - This class provides setter/getter methods to set/get the required field.

4. **Response**
   - Response class is used whenever a peer server in the network wants to send response for the request received. This class has three fields – **responseCode, responseData** and **otherData**.
   - Whenever a peer server wants to send a response, it creates a Response object, sets it **responseCode** (For example, 200 if request was processed successfully) and **responseData** to appropriate data as requested and sends the object using **ObjectOutputStream**. Sometimes, peer server sets data in the field **otherData**. This is used once in the system for the replication mechanism.
   - A peer client tracks incoming Response object using **ObjectInputStream**.

- This class provides setter/getter methods to set/get the required field.

5. **IPAddressValidator**
   - This class is used for validating an IP address. It uses regular expression and pattern matching to check whether the provided IP address in the configuration file is valid.

6. **NetworkUtility**
   - This class provides a method – getLocalAddress() which gives the IP Address of the Peer. This is used to check whether the selected peer (through hash function) for <key, value> pair insertion is itself or not. In case it is itself, then we don't need to create a socket connection.

7. **DistributedHashTable**
   - DistributedHashTable is the main class of the system which runs when the program is started using "make run" command.
   - This class loads the configuration file. It retrieves the IP address of all the peers which will be in network and also IP address of the peers who will be acting as replication servers. A normal peer can also act as replication peer if its IP address is mentioned in "REPLICATION_NODES" field.
   - After loading the configuration file, it starts an instance of peer client on a single separate thread and as a peer server listens to port 20000. Any peer client who wants to connect to this peer can connect using the peer's IP address and Port 20000. I have used port 20000 for my Distributed Hash Table system.
   - This class also defines the most important data structures of this system like **hashTable**, **replicatedHashTable**, **networkMap** and **replicationNodes**.
   - **hashTable** is a concurrent hash map (Java's **ConcurrentHashMap**) which stores all the <Key, Value> pair of its own peer. I have concurrent hash map because it is thread safe i.e. multiple threads can perform read/write operations on the **hashTable** without any kind of deadlock or inconsistency.
   - **replicatedHashTable** is also a concurrent hash map data structure which stores the **hashTable** i.e. <Key, Value> pairs of all the peers in the network. It maps the **hashTable** of a peer by the peer's IP address. This data structure is also thread safe.
   - **networkMap** is a hash map structure which stores the Peer ID and IP Address of all the peers in the network. I haven't used concurrent hash map in this case because multiple threads will only read this data and not modify it.
   - **replicationNodes** is a list which stores the IP Address of all the peers who are responsible for storing the **hashTable** of all the peers in the network.

8. **PeerClient**
   - The Peer Client provides following functionalities – PUT, GET, DELETE, PRINT PEER SERVER LOG and EXIT. The Peer Client requires the Host Address (IP address) of the Peer it wants to connect so. It can connect to other peer using a socket connection using that host address and port 20000.
   - A user can add (PUT) a <Key, Value> pair in the Distributed Hash Table (DHT). The peer client asks user for the key and value and then connects to the appropriate peer by hashing the key and sends a request to that peer to store the <Key, Value> pair in its **hashTable**. The hash function is explained later in this document. The peer client calls **put(key, value)** function to add the value.

The put(key, value) function returns true if the key is successfully added in the DHT else returns false.

- A user can search (GET) for a <Key, Value> pair in DHT using Key. The peer client asks user for the key, performs hash function on the input Key and connects to the appropriate peer to request the Value for the input Key. The peer client calls **get(key)** function to get the Value for the Key. The get(key) function returns the Value of the Key if the Key is present in the DHT else it returns null (nothing).
- If the peer client is unable to connect to the respective peer (may be the peer is down), then it requests the replication nodes to check for the Value of the input Key. It calls **searchReplica(key)** in this case.
- A user can remove (DELETE) a <Key, Value> pair from the DHT using Key. The peer client asks user for the key, performs hash function on the input Key and connects to the appropriate peer to delete the <Key, Value> pair for the input Key. The peer client calls **delete(key)** function to delete the <Key, Value> pair. The delete(key) function returns the true if the <Key, Value> pair was successfully deleted from the DHT else returns false. In case the Key is not present, then too it returns true.
- The peer client on initialization (on startup) retrieves its **hashTable** (if any) from the replication nodes by calling **retrieveHashTable().** Since I have implemented replication mechanism, this is necessary because if the peer shuts down for some reason and starts again, it should have its own data. If this was implemented then, if the peer starts after shutting down, then it won't have its data while the replication nodes will have peer's data thereby creating inconsistency between the peer's hash table and replication hash table.
- The peer client on initialization (on startup) also retrieves its replication hash table from other replication nodes only if it is amongst the replication nodes in the network. If it is the only replication node in the network, then this won't work. It calls **retrieveReplicationHashTable**() for this purpose.
- Finally, it provides functions to validate Key and Value. Key must be not more than 24 bytes (12 Java characters) and Value must not be more than 1000 bytes (500 Java characters).

9. PeerServer

- The Peer Server listens to the requests by other Peer Clients in the network. It serves requests like PUT, GET, DELETE, etc.
- If the request is **PUT** i.e. to add a <Key, Value> pair, then first it checks its **hashTable** if that Key already exists. If the Key exists it sends a **Response** object with **responseCode** = 300 which means that Key is already present. If the peer client decides to overwrite the value then it sends a **Request** object with **requestType** = "PUT_FORCE". In this case, the peer server directly adds a value. If present, it is overwritten. The old Value for that Key is replaced by the new Value. If the <Key, Value> pair is successfully added then the peer server sends a **Response** object with **requestType** = 200.
- If the request is **GET** i.e. to search and send the Value for the requested Key, then the peer server checks its **hashTable** and sends a **Response** object with **requestType** = 200 and **requestData** = Value if the Value for the requested Key exists else it sends a **Response** object with **requestType** = 404 meaning Value not found for the specified Key.

- If the request is **DELETE** i.e. to remove/delete the <Key, Value> pair of the requested Key, then the peer server deletes the <Key, Value> pair of that Key from its **hashTable** and sends a **Response** object with **requestType** = 200.
- The requests **R_PUT**, **R_GET** and **R_DELETE** is handled same way as requests PUT, GET and DELETE respectively except the operations are done on **replicatedHashTable** rather than **hashTable**.
- If request = **GET_HASHTABLE**, then it sends the **hashTable** (from **replicatedHashTable**) of the peer client who has made the request. For response, it sends a **Response** object with **responseType** = 200 and **otherData** = **hashTable**. In case, there is no hash table for the requestor peer, then **otherData** = null.
- If request = **GET_R_HASHTABLE**, then it sends the **replicatedHashTable** as is to the peer client who has made the request. For response, it sends a **Response** object with **responseType** = 200 and **otherData** = **replicatedHashTable**.

10. ReplicationService

- The ReplicationService class provides data replication services used by the Peer Server.
- If it receives a PUT <Key, Value> request, then it sends a R_PUT <Key, Value> request to all the peers contained in the **replicationNodes**. So, it basically tells all the replication nodes to store a replica of this <Key, Value> pair.
- If it receives a DELETE <Key> request, then it sends a R_DELETE <Key> request to all the peers contained in the **replicationNodes**. So, it basically tells all the replication nodes to delete the <Key, Value> pair with the specified Key.

11. Data Replication

- You might have got a basic idea about data (hash table) replication implemented by this DHT system.
- When the Peer Server receives a PUT <Key, Value> request, after serving the request it calls the **ReplicationService** on a different thread with PUT <Key, Value> request. The **ReplicationService** sends a R_PUT <Key, Value> request to all the replication nodes (found through **replicationNodes** in configuration file). The Peer Server of all the replication nodes serves the R_PUT <Key, Value> request by adding the <Key, Value> pair in its **replicatedHashTable** structure. In this way, whenever a peer in the network adds a <Key, Value> pair in DHT, it is also stored in all the replication nodes.
- When the Peer Server receives a DELETE <Key> request, after serving the request it calls the **ReplicationService** on a different thread with DELETE <Key > request. The **ReplicationService** sends a R_DELETE <Key > request to all the replication nodes (found through **replicationNodes** in configuration file). The Peer Server of all the replication nodes serves the R_PUT <Key > request by removing/deleting the <Key, Value> pair having the specified Key from its **replicatedHashTable** structure. In this way, whenever a peer in the network deletes a <Key, Value> pair in DHT, it is also deleted from all the replication nodes.
- As mentioned earlier, if any peer client in the network is not able to connect to another peer for GET operation i.e. retrieving the Value for a Key, then it calls searchReplica(key) which in turn sends a R_GET (Key) request to all the peers in **replicationNodes**. In this scenario, the Peer Servers, server the R_GET <Key> request by retrieving the Value for the Key from its **replicatedHashTable**. In this way, the Value for the Key is retrieved from the replication nodes in case the original nodes are not active.

- Note that in order for data replication to work, at least one of the data replication nodes must be started first. In case this is not done, then the data replication node will store a replica of only those <Key, Value> pairs which are added after the initialization of the replication node.
- Also, since this system does not load the peers in the network dynamically (i.e. a peer isn't added after all the peers in the configuration file are initialized), if a node decides to act as replication node after it is initialized, it won't be able to do so. There is one way to do this – the node (peer) will restart itself. This will work because, after starting, the peer will contact other replication nodes to get its data and also to retrieve the replication data. But, again this won't work in case this is the first replication peer which is made the replication node after starting the program.

12. LogUtility

- This class provides functionality of creating and modifying log files which is used by the Peer Server.

Suggested improvements in the current system:

- No authentication is done between peers. If a system gets to know that this Distributed Hash Table system works on port 20000 can exploit the hash tables in the network. Authentication mechanism is required amongst the peers so that only those peers are connected to a peer if the requestor (peer) is authorized and authenticated with the other peer.

  This can be done using simple user id and password mechanism. The peer will need to have a User ID and Password to connect to the other peer. For better security, instead of passing raw passwords trough the network, the peer will calculate hash of the password and send the hash value (i.e. use SHA algorithm).
- The code of the programs can be improved so as to work efficiently. In a network of 5-10 nodes, the difference may not be visible but in a distributed network of more than 10 nodes the difference may be important.
- Data replication can be made more efficient by using various parameters like peer's network speed, reliability, how often the peer remains connected/disconnected. These parameters can be used and algorithm can be devised which selects a better replication node dynamically amongst the connected peers.
- A better Hash Table structure can be implemented for faster PUT, GET and DELETE operations.