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 t=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. **Request**
   - Request class is used whenever a node wants to request something from another node/server. This class has two fields - **requestType** and **requestData**.
   - Whenever a node/server wants to request something, it creates a Request object, sets it **requestType** which is basically a command like REGISTER, UNREGISTER, LOOKUP, DISCONNECT, etc. and sets the **requestData** to appropriate data it wants to send while requesting something and sends the object using **ObjectOutputStream**.
   - A node tracks incoming Request object using **ObjectInputStream**.
   - This class provides setter/getter methods to set/get the required field.

2. **Response**
   - Response class is used whenever a node wants to send response for the request received. This class has two fields – **responseData** and **responseCode**.
   - Whenever a node/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**.
   - A node tracks incoming Request object using **ObjectInputStream**.
   - This class provides setter/getter methods to set/get the required field.

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

4. **FileUtility**
   - This class is built to work specifically on files mostly. This class is mostly used while transferring files amongst nodes.
   - The methods of this class are –
   - *getFiles(path)* – This method retrieves all the files from the specified path of the file system. It is used when a peer specifies a path whose files are to be registered with the Indexing Server.
   - *getFileLocation(filename, locations)* – This method searches for the specified file using the specified **fileName** amongst the given **locations**. When a peer receives a request to transfer a file to another peer, it uses this method to search the file from its all the registered file locations.
   - *downloadFile(hostAddress, port, filename)* – A peer uses this method when it wants to retrieve/download a file from another peer. It specifies the IP Address and Port of the peer from which it wants to download the file. In my system, I have used port 20000 for file transfer purpose i.e. Peer when it is acting as server listens for file transfer request using port 20000. In this function, we use Socket API to establish a connection to the specified peer (host) and then request to send the file and we receive and save it.

- *replicateFile(hostAddress, port, filename)* – This method functions same as downloadFile(…) except that it doesn't print anything on console. It is used by the replication node to get the files from all the nodes and save a copy of those files.
- *printFile(filename)* – This method is used by the Peer to print the downloaded text file. It prints the first 1000 characters of the text file.

5. IndexingServer

- The purpose of Indexing Server is to maintain a list of peers connected to the server and also maintain a list of all the files that are registered by the peer i.e. list of files that the peer wants to make available for file sharing/transfer. I am using a **ConcurrentHashMap** for this purpose because Java's concurrent hash map is serializable as well as thread safe. It is important to use a thread safe mechanism because there will be multiple threads accessing this data simultaneously.
- ConcurrentHashMap stores data in form of key value pairs. I have kept the key in following format – Peer_id#Peer_ip#HHmmss and the value is list of files registered by the respective peer. (HHmmss is time (hours minutes seconds) and I have used it to just to keep the key unique. I call this structure as indexing datatbase.
- The Indexing Server is also storing a list if all the peers (nodes) who are acting as replication nodes i.e. they are keeping a copy of all the registered files in the network.
- I have also maintained a list of all the locations (paths) whose files are registered by the peers. This is because, let's say a peer was down for some time and then it initializes again, then all the local variables of that program would have lost. So, when the peer connects to the Indexing Server, we check if that peer had registered any files previously. If it had, then we send him the list if the locations (paths) whose files were registered so that he can serve other peers for file transfer.
- When Indexing Server program is run, it keeps on listening to port 10000. Any peer who is trying to connect the Indexing Server using server's IP address and port 10000, all its request is served by the Indexing Server. So, whenever the server detects a new incoming connection, it creates a new thread and dedicates that thread to that connection (i.e. that peer). Hence, one thread is for one peer and as many peers connect, that many threads are created.
- When the connection is established with the peer, the server sends a welcoming message to the peer and asks if it wants to serve as replication node.
- If the peer replies yes, the server adds that peer's address to the list of replication nodes and responds him with the list of files registered by all the peers connected to the server. Then the peer requests all the peers (as per the list provided by the server) to send their registered files to him so that it can keep a replica of those files.
- After all this is done, the server runs on an infinite loop so that it can continuously listen for all the requests sent by the peer. The various requests that are served by the Indexing Server are – REGISTER, LOOKUP, UNREGISTER, GET_BACKUP_NODES and DISCONNECT.
- When the server receives a **REGISTER** request, the list of files is present in the **requestData** field. So, we grab that data and add it to the indexing database which is a concurrent hash map structure. Also, we send a **REPLICATE_DATA** request with the newly registered files data to all the peers who are acting as replication nodes (As mentioned above, we store list of nodes serving as replicator) so that they can contact peer and retrieve a copy and store it with them as well. If everything goes well, we respond to the peer by sending **responseCode** = 200.

- When the server receives a **LOOKUP** request, the name of the file to be searched is present in the **requestData** field. We search that file in our indexing database. We look all entries of the indexing database (even if we found the peer in the first entry) because we want to send details of all the peers who have that file. If the file was found in our indexing database, we respond to the peer by sending **responseCode** = 200 and **responseData** = List of peers (Peer ID and Peer IP Address) using **Response** object. If not found then, we respond to the peer by sending **responseCode** = 200 and **responseData** = File not found.
- The server receives a **UNREGISTER** request when a peer wants to stop sharing its files in the file transfer network. We simply remove all the entries of that peer from our indexing database and send a **DELETE_DATA** request to all the peers who are acting as replication nodes so that they can delete the copies of all those files which have been unregistered. If everything goes well **responseCode** = 200 else **responseCode** = 400
- The server may receive **GET_BACKUP_NODES** from one of its connected peers when the peer is not able to download/retrieve the file from the original owner (peer) of that file (may be because the peer was shutdown temporarily). In this case, the server responds with list of replication nodes so that the peer can download the file from the replication nodes. Note that the user who is requesting the file doesn't even come to know that the file was served to him from a different peer than what he/she may have requested. This is one of the benefit of data replication.
- If a server receives a **DISCONNECT** request, the server closes the socket connection to that peer and interrupts its own thread because he was serving that peer.

6. Peer

- The Peer serves to purpose – acts as client when communicating with indexing server or requesting a file from another peer and acts as server while listening/serving file transfer requests of other peers in the network.
- I have maintained a list of all the indexed locations of my peer i.e. path of all the files which has been registered with the indexing server. The list is kept thread safe because multiple threads may access that data in parallel.
- When the Peer program is run, a new thread is created which works a Peer client who communicates with the indexing server. Also, the Peer server part of the program keeps on listening to port 20000. Any peer who wants to request a file from another peer will connect to the respective peer using its IP address and Port 20000. I have used port 20000 for file transfer mechanism.
- So, whenever it receives an incoming connection request (on port 20000) for file request, it creates a new thread to serve that peer and complete the request. So, as many file transfer requests come, that many threads are created.
- Whenever the Peer Server receives a **DOWNLOAD** request which is basically a file transfer request i.e. another peer is requesting a file, the name of the file is present in the **requestData** field. The Peer Server then searches the path of the file from its own list of indexed locations (as mentioned above) to get the full qualified path of the file. Note that we are just storing the name of the files and not the path in the Indexing Server's indexing database due to security reasons. It is not good to reveal path structure to other nodes in a distributed network. After getting the full path, the Peer Server transfers the file in stream and Peer Client accepts the file in chunks because of the limited buffer size. Note that we can't send/receive a 10 MB file in one shot. It has to be done

chunk by chunk in stream. The Peer Client then builds the file from the data received in the stream and stores the files in downloads folder.

- The Peer Client provides following functionalities – REGISTER, LOOKUP, UNREGISTER, PRINT DOWNLAOD LOG and DISCONNECT. The Peer Client requires the Host Address (IP address) of the Indexing Server so that it can create a socket connection using that host address and port 10000.
- When the peer wants to register its files with the indexing server, it asks the user for the path from where the files are to be scanned. Peer uses FileUtlity's getFiles(…) method to get the name of all the files in the given path. Then, it creates a request object having **requestType** = **REGISTER** and **requestData** = List of file(s) to be registered and sends the request object to the Indexing Server using the socket connection. If it receives **responseCode** = 200, then files were registered successfully.
- The user can search a file by providing the filename. For lookup, the peer creates a request object having **requestType** = **LOOKUP** and **requestData** = file name to be searched and sends the request object to the Indexing Server using the socket connection. If the peer receives **responseCode** = 200, then file was found and list of peers where the file is present is available in the **responseData** field. If the file was found, the peer client provides the user the option to download the file and if it's a text file it provides the option to download as well as print the file. If there multiple peers available for the file, then user can select the peer to use for downloading the file using Peer ID. If the file was not found, **responseCode** = 404.
- This system works on all kind of files. I have tested this system on .txt, .jpg, .docx, .zip, .pdf files.
- The user can unregister all its files if required. This may be required if the user no longer wants to share those files or user is turning off the peer. The peer does this by sending a request object with **requestType = UNREGISTER.** The Indexing server then unregisters i.e. removes the entries of all the files of that peer. If everything goes well, the peer receives responseCode = 200.
- When the user exits the system, the peer sends a **DISCONNECT** request to the indexing server so that the thread serving that peer is interrupted and closes the socket connection with the server.

7. Data Replication

- Some of the data replication design is already covered while explaining the Indexing Server design. I have designed the data replication module by re-using the components and mechanism used for file transfer.
- Any peer can act as a replication node. Therefore, the Peer Server also acts as a listener for replication related requests.
- If you remember, as mentioned above, the Indexing Server asks peer if they want to act as replication node. If the peer agrees to act as replication node then, an inner class called **ReplicationService** is run on a different thread. The same class is also run when the Peer Server receives a **REPLICATE_DATA** request.
- The primary and only task of Replication Service is to iterate through all the files sent to them by the Peer Client/Server (Indexing Server sends to Peer Client/Server) and send a request to all those peers to transfer their registered files to the Replication peer i.e. to the requestor. Once all these files are replicated, this thread interrupts itself.
- The Peer Server may receive a **REPLICATE_DATA** request from the Indexing Server whenever a peer in the network registers its new files with the indexing server. The list of new files is present in the **requestData**. The Peer Server then creates a new replication service thread which in turn contacts the peer to transfer the newly registered files to himself. Receives

- The Peer Server receives a DELETE_DATA request from the Indexing Server whenever a peer in the network unregisters its files from the Indexing Server. The Peer Server then deletes all the files of that respective peer so as free up some space.

8. LogUtility

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

Suggested improvements in the current system:

- Data fields in the Request and Response class can be increased so that multiple kind of data can be sent in a single request/response.
- This system doesn't do any kind of error checking at application level after the file transfer is done. The transferred file may become corrupt under certain conditions. (The files transfer is perfect in the current system. Just that it's better to add an extra error checking mechanism.)

  This can be done using checksum mechanism. Just before sending the file, the node will send a Response object containing details about file including the checksum. After the file transfer is complete, the peer can compare the received checksum with the downloaded file's checksum. If they are same, then file was transferred correctly else not.
- No authentication is done when the peer connects to the server in this system. Authentication mechanism so that the Indexing Server is able to authenticate the peer and accept only authorized connections. Also, while serving the file transfer request, the peer should check if the requestor (peer) is authorized and authenticated with the Indexing Server.

  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 Indexing Server. 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). Also, the indexing server will maintain a list of connected and authenticated users. So, before initiating file transfer request, the peer server will check whether the peer is connected and authenticated with the Indexing Server. If yes then transfer file else don't.
- The code of the programs can be improved so as to work efficiently. In a network of 10-20 nodes, the difference may not be visible but in a distributed network of more than 100 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 amongst the connected peers.