Design Document

A Simple Decentralized Peer to Peer File Sharing System

Introduction:

Peer to peer file sharing system allows computer systems to share files by communicating with other systems connected in a peer to peer network. The backbone of a peer to peer system is the Distributed Hash Table (DHT). The files in each peer can be sent to / received from the peer by checking the DHT consistently and routing the messages accordingly to the required peer. Each peer act as both client and server and has its own DHT. The messages communicated between different peers can have several delimiter, which may include files, functions such as read/write and the portion of message can be hashed to find the required peer. The Design document covers the designing of a decentralized peer to peer file sharing system, the tradeoffs that were made in designing the program, possible improvements that can be implemented in the future and performance evaluation.

Pre-requisite:

- Linux Operating System
- Java Installed in system. I installed open-jdk 1.7 version. Java can be found in /usr/lib/jvm.

Concepts that were used in the program:

- Socket programming - For communication
- Multithreading - For handling multiple client-server connections.

Design of the Decentralized Peer to Peer file sharing system:

The program contains the following java files which are discussed as follows.

1) Peer class

The peer.java contains the main method and it is the starting point of the application. It uses command line arguments to process information from the client side and encapsulates the information in the peer-node object and passes the information to server thread.

The various arguments that can be given from the client side are peer' Id, port, hostnames etc.

2) Peer-node class

It contains the required properties of all the peer nodes connected in a network. It creates an abstraction for request and response format of the messages to be used in connection with the other peers. The Distributed Hash table is implemented in this class , which is used to transfer the message to the required server thread.

3) Server-Thread class

The server-threaded class uses multithreading approach to serve multiple clients and process the requests from the peers.

4) Request class

This class sends the client's message to the server for processing the message. The message may contain functions for put (write) / get (read).

5) Response class

This class send the processed message from the server side to the client.

6) Settings class

The protocol version of the message is defined in this class. It act as message style properties which can be used as settings.

7) Indexing-Server class

The Indexing Server holds a list of filenames and ip/port of the peer. It uses hash-map to store filenames and ip/port as key-value pairs. It continuously listens to all the peers. Whenever each peer calls its Indexing-Server class, it internally calls the peer-Thread file, where the information stored in the hash-map is encapsulated and sent to peer-node class where the messages are distributed and sent to the required server thread.

8) peerClient class

The peerClient file of a peer talks with other peerClient file to obtain the required file through byte streaming operations. Each peerClient file act as both server and client to communicate with each other.

I implemented the DHT for 8 peers running concurrently in a peer to peer network.

Design trade-off:

The program is designed in such a way that it will run in a single Virtual Machine (VM). Though the program can be scalable across multiple VM's by doing small modifications in the code, it is highly necessary to configure the network interfaces of each VM's in order to run the peer to peer Distributed Hash Table program successfully. I kept the working of my program simple by avoiding the network interface complexities, thus sacrificing scalability. In my program, I kept the threading part closer to the server side, rather than near the client. Though introducing threading within the client program for faster file sharing process will balance the load between multiple peers, synchronization cost will be high for locking and unlocking the threads. Also I kept the byte size as fixed 1024 bytes for your testing purposes. This holds good for file transfer of smaller size. Although increasing the byte size will make streaming of larger file size possible, but it will make full use of I/O. When multiple peer try to talk each other at the same time, they need to significantly wait for a large amount of time to acquire its share of I/O.

Improvements and extension:

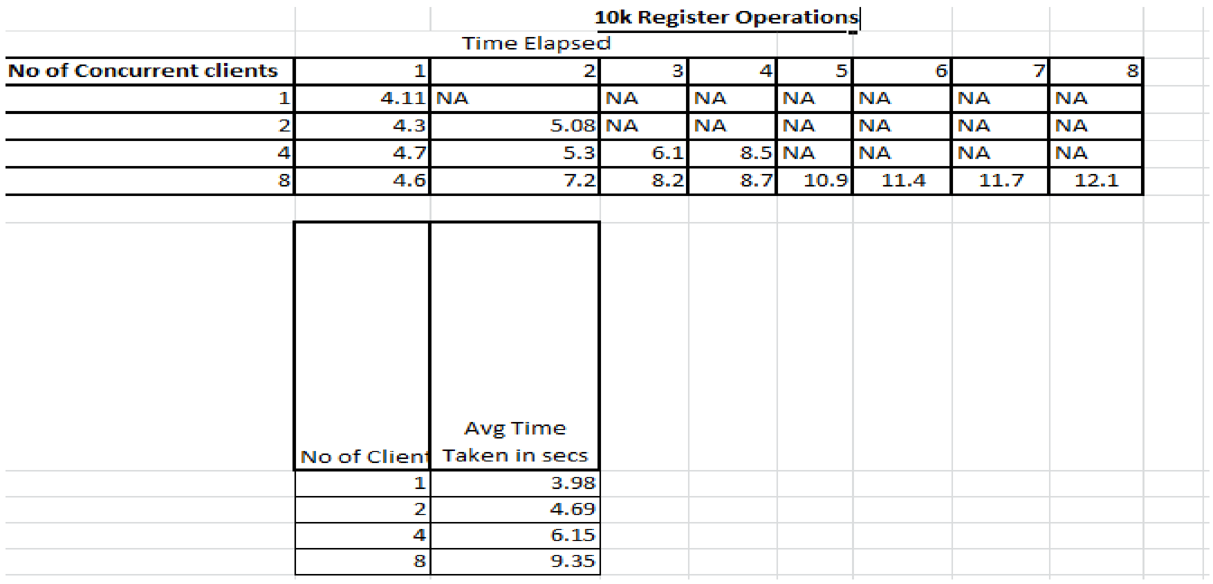
- The peer to peer decentralized file sharing progam can be scalable across multiple vm's / operating systems by modifying the network interfaces (NAT, Bridged etc) to support socket communication. For example in each peer program across multiple vm's, looping around various network interface adapters other than local loopback IP will permit the socket communication across machines. Network firewall should also be taken into consideration during peer to peer communication.
- The hash functions can be made even more complex by implementing complex mathematical algorithms to hash the key
- The peers forming the peer to peer network can have a unique password shared across a peer-group , that needs to be used for joining the network, so that unauthorized peers cannot join the network. Also the hashing function should be hided from unauthorized peers accessing the network, such that any messages of suspicious activity is not easily sent into a peer to peer network.
- The replication strategy can be made more effective by sending periodic heartbeat signals to all other peers, such that it can be an pro-active measure to check whether all peers are alive or dead. If any peer does not send any heartbeat signal to other peers at some point of time, it is assumed like that peer is dead, and the replication can be done automatically to other peers using a configuration file having some replication factor.
- I will implement the peer to peer decentralized file sharing system on Amazon web services by creating an EC2 instance in the future, such that the performance can be more closely analyzed than testing in the localized system.

Performance Evaluation:

I have performed the evaluation on 10K register (put) operations and 10K query (get) operation and 10K (lookup) operations on all the client and found that , as the number of peers are operating concurrently, there is a slight delay in the time for performing the lookup and get, since multiple peers may rehash the key stored in its Distributed hash table. For my local testing I included timers in the log file, to check the time for 10K register, 10K lookup and 10K get operation. The average time taken for put operation is generally less than, the time taken for get operation.

Experiment 1

a) Elapsed Time (in Seconds) for 10K register operations.

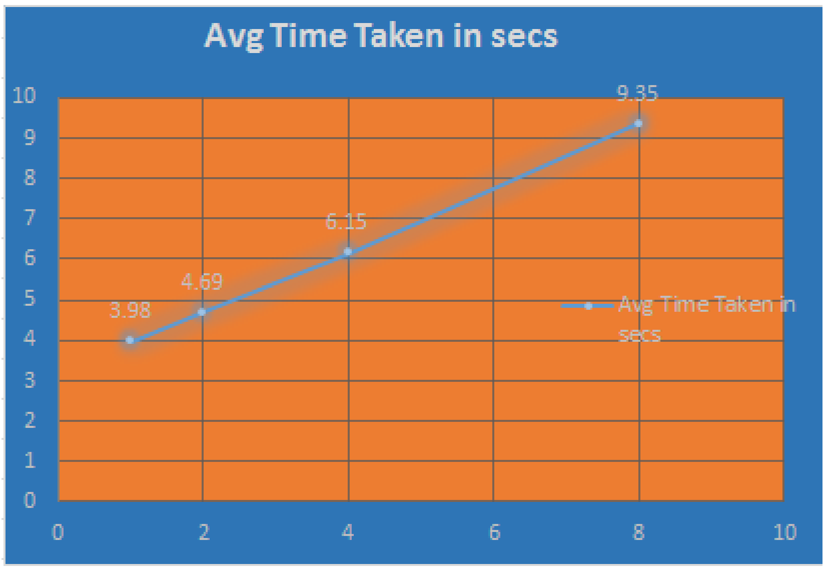


10k Register Operations

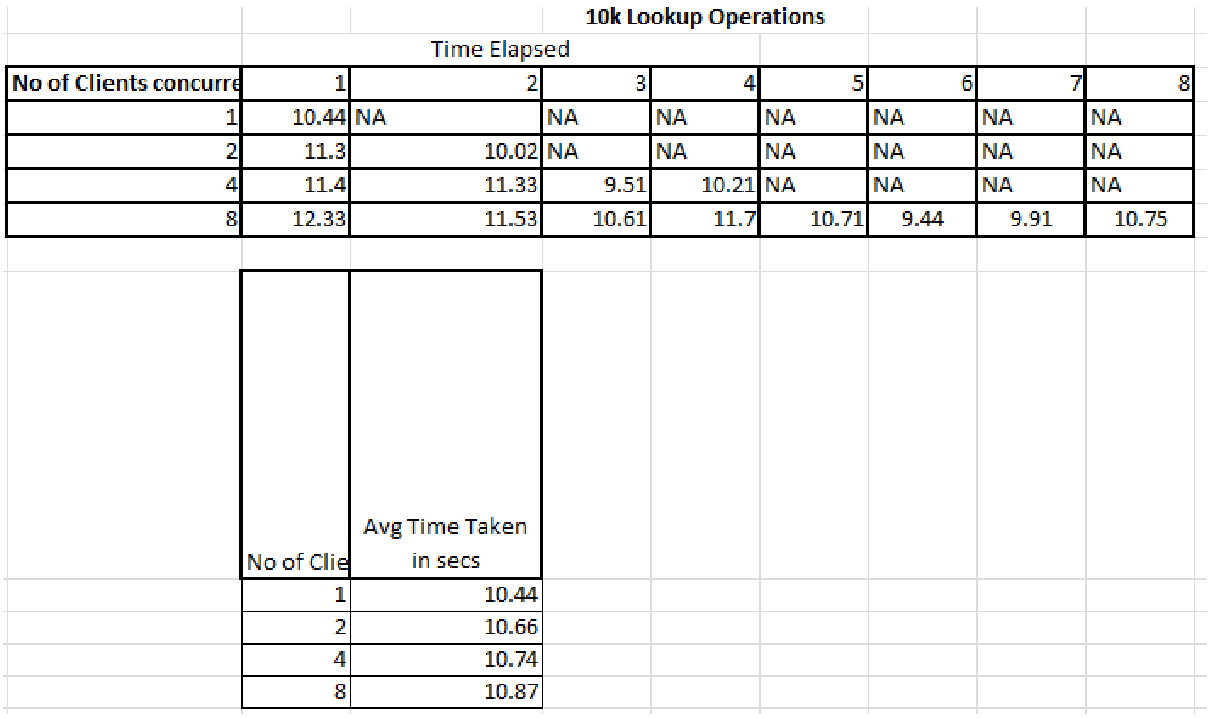
Time Elapsed

No of Concurrent clients	1	2	3	4	5	6	7	8
1	4.11	NA	NA	NA	NA	NA	NA	NA
2	4.3	5.08	NA	NA	NA	NA	NA	NA
4	4.7	5.3	6.1	8.5	NA	NA	NA	NA
8	4.6	7.2	8.2	8.7	10.9	11.4	11.7	12.1

No of Client	Avg Time Taken in secs
1	3.98
2	4.69
4	6.15
8	9.35



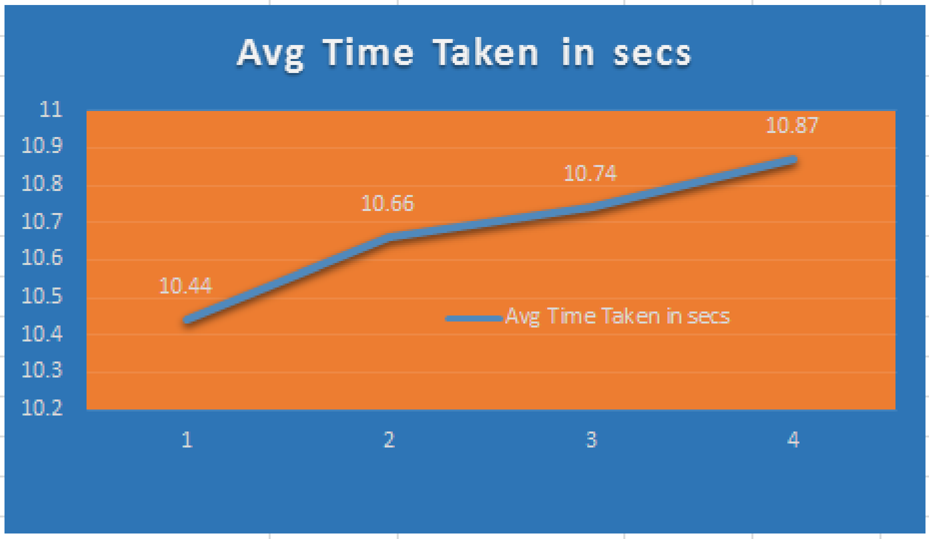
b) Elapsed Time (in Seconds) for 10K lookup operations.



10k Lookup Operations

	Time Elapsed							
No of Clients concurre	1	2	3	4	5	6	7	8
1	10.44	NA	NA	NA	NA	NA	NA	NA
2	11.3	10.02	NA	NA	NA	NA	NA	NA
4	11.4	11.33	9.51	10.21	NA	NA	NA	NA
8	12.33	11.53	10.61	11.7	10.71	9.44	9.91	10.75

No of Clie	Avg Time Taken in secs
1	10.44
2	10.66
4	10.74
8	10.87



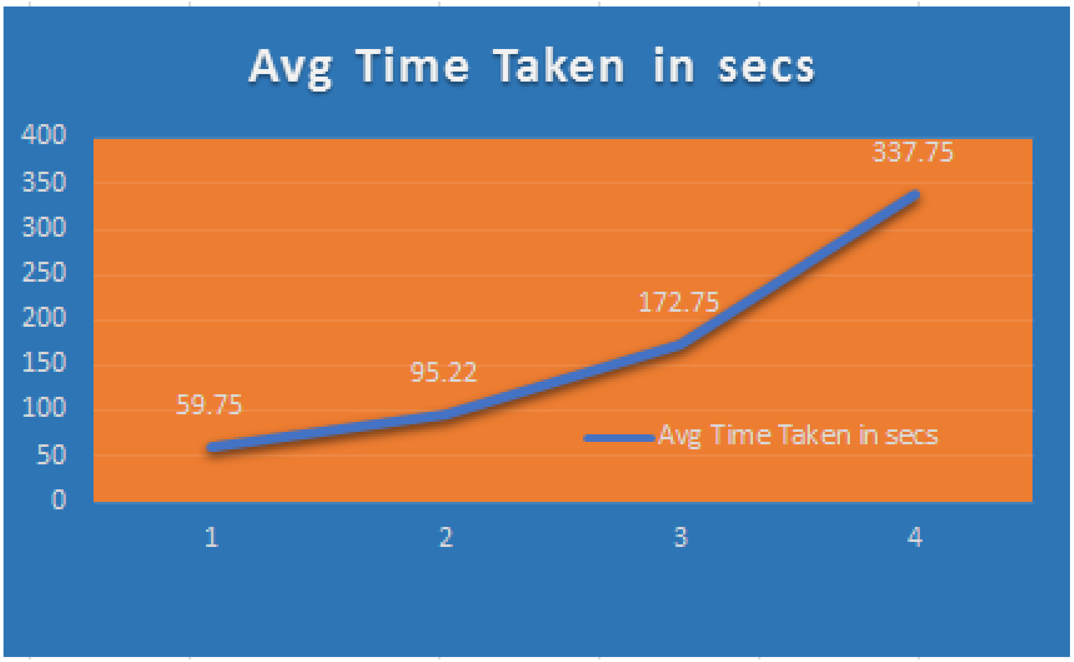
c) Elapsed Time (In Seconds) for 10K get operations

10k Get Operations

Time Elapsed

No of Clie	1	2	3	4	5	6	7	8
1	59.75	NA	NA	NA	NA	NA	NA	NA
2	86.21	104.23	NA	NA	NA	NA	NA	NA
4	185	161	155	190	NA	NA	NA	NA
8	241	312	317	350	373	369	350	390

No of Clie	Avg Time Taken in secs
1	59.75
2	95.22
4	172.75
8	337.75



Experiment 2

I have performed the analysis with various file sizes locally (KB, MB, GB) for file sharing between eight concurrent clients and found that as the file size for sharing is increased, the time taken to download is increased greatly.

File Size	Time taken to Download (in Milliseconds)
1KB	140
10KB	641
100KB	1265
1MB	6222
10MB	12521
100MB	97012
1GB	741236

Achieved throughput = (Total bytes transferred in the network) / (Total time taken)

Achieved Throughput is approximately 1 MB / sec.

Based on comparing the centralized and decentralized peer to peer systems, both has some advantages and disadvantages.

For example in Centralized system, if the centralized indexing server is down or killed, the whole metadata is lost and the peer to peer system will be crashed.

In centralized system, the communication costs for lookup and get operations is very less compared to the decentralized system, since there is less contention for network bandwidth as compared to the decentralized system. The centralized system is easy to maintain compared to the decentralized system.

In Decentralized system, if one of the server is down, the peer replicates the key value pairs of the messages to other peers, such that the metadata containing the file information is recreated in other peers. The hashing mechanism in decentralized system should be very strong, such that unauthorized users cannot break into the peer to peer network easily. Load can be effectively balanced in a decentralized system as the load can be made to share evenly across servers.

Conclusion:

The Decentralized peer to peer file sharing application was successfully implemented as per the guidelines. I am confident that, I could add some more features as improvements to the existing system, so that I could improve its efficiency. Last but not the least I thank Dr. Ion Raicu and the respective Teaching Assistants for guiding me during any clarifications or doubts.