Written Submission: Project 1

Student Name: Aoshuai Liu
Student ID 1146839

<u>Question 1:</u>

→My P2P downloading strategy involves pre-sorting of the available resources before deciding which Peer to download from.

1. Any downloading process starts with a corresponding LookUpRequest with the index server to obtain all the sharing peers that provide uploading services to such file hash (i.e., provided matching fileMd5 is essentially matching the content of the file).
2. Then, we will delegate a EchoDownloader (derived class of Thread) to process the hitting IndexElements by sending a DownloadRequest(message) all the sharers and expecting to receive a DownloadReply message that contains their sharing status (i.e., Seed or P2P Downloading) and their block availabilities.
3. EchoDownloader will prioritise our BlockRequests based on sharers' block availability, and sort seeds to maximise the efficiency of multiple-request per P2P connection. That is, any "Seeding" sharers (i.e., sharing a completely downloaded file) will be used first and the rest of "P2P Downloading" sharers (i.e., concurrently downloading such file while uploading blocks) for in a descending order following the "Seeding sharers" as back up.
4. Before Downloading blocks, we will notify the index server as the downloading peer itself is concurrently sharing such file as well (i.e., enabling the scalability advantage of P2P architecture).
5. Then we will loop through these sorted seed and establish block transmission with the sharer's uploading thread provided that our downloading FileMgr verify the correctness of block index and block bytes.

→Note, downloading order implemented here is to be natural as the management concurrent downloading and uploading will be more effective. That is, keeping tracks of their sequential downloading block allow this downloading peer to concurrently serve the blockRequests from other peer.

- For example, if this peer has concurrently uploaded block 0 – 2 with others while it is downloading block 3, other downloaders will be responded with a BlockInProgress message to notify them that his peer is still downloading such block.
- These Threaded downloaders will sleep for an appropriate time to retry for block 3 as they know the Sharer is most likely downloading this block (unless sharer really had issue on such block, then at most downloaders will try 3 times) since all peers are implementing sequential downloading order.
- Thus, we will reduce the ineffective loops for block downloading as we generally only loop through each sharer's block once (this is important when the number of blocks is large), with the leverage of 3 error tolerance per block, we can maximise the blocks transmission from one sharer instead of blindly missing such sharer's block resources.

→On the other hand, this advantage may become efficiency issues if sharer tends to apply a large block-size partition that each block may takes an exceedingly long time to download and sharing with others. In these situations, natural order downloading will decrease the efficiency in P2P file transmission.

→This is why allowing peers to downloading blocks with arbitrary indexes (or even random) can mitigate this problem when there are enough Peers within the system. Knowing that every block is potentially completed by some peers, upcoming downloaders can choose to switch to download any available block that is shared by that sharer and moving on to others to quickly build up the whole file.

Question 2:

→For Peer and Index Server communication, some consistent functionalities can be implemented to reduce the overheads in sharing record management.

1. The IndexMgr merely perform the sharing Indexelement of a single request file, but the consistency of sharer's secret / sharer's port is not managed by the server. If the peer is config to undertake different sharing secret or port, the previously existing sharing record will not be managed and any P2P downloading will be refused or throwing Exceptions due to out-of-date information.

- As I implemented in Peer.java, applySecretConsistency() method, this situation invokes inefficient overheads in scaling distributed systems needs to calling an handshake for every updating sharing IndexElement. Instead, we can either create a new message class that notifies the Server our secret has been config to certain value or implementing a new boolean field for the server to identify any requirement of updating all sharing records' secret.

2. Similarly, the shutdown procedure of a peer requires a iterative calling of handshaking with the index server to perform DropRequest on each single existing sharing indexElement. Thus, we can implement a new Quitting message to notify the index server and indirectly all peers that we are no longer sharing any records with a single connection with idxsrv.

- Moreover, this invokes another issue on peers' ungraceful quitting of P2P sharing, which index server does not detect any disconnected peers that previously updated the sharing files. This introduces inefficient connection establishment for other peers that performed lookups on these files as the per have quitted already. Therefore, can as server to create a new thread with the IndexMgr to send an IsAlive() Message after every discrete sleep() period to reconfirm the sharing peers is still online.

→For Peer-to-Peer transmission, great number of overheads occurs at the pre-sorting phase where we are not efficiently exploiting the multiple-request per connection structure of peer services. Provided with large hitting IndexElements, we have loop through every sharer to establish communication with them (handshake and termination) to obtain a DownloadReply message that contains the sharing status ("Seeding" and "P2P Downloading") and block Availability needed to descendingly sort these seeds.

- This can be avoided if we add these two fields into the sharing indexing element when we passed it to the index server, we no longer need any DownloadRequest / DownloadReply communications to sort out the seeds we have. Thus, the only communication left with other peer's Echo Servers are just unbounded sequences of block transmissions, which fully take advantage of the multiple-request per connection setting.

→Note, in order to update any record of them, these implementations are not backward compatibility as this functionality require a reconstruction of the sharerMap to identify all IndexElement of such sharer, and any existing indexElement will not provide those fields we suggested. Similarly, so as the peer's application where the information it needs to request to share (so as consistency maintenance) and program existing will need to change as well.

Question 3:

→Implemented by both the index server and the Peer Service, the socket timeout parameter are passed to their IO Thread to manage incoming connections from peers (client), which calls setSoTimeout() method to specify a timer for all the IO methods of such socket to prevent the indefinitely waiting time due to unexcepted connection issues on either side of the transmission.

- In this case, it is the setting timeout that a server (i.e. reading request) or a client (i.e. reading Reponses) will block for its methods on reading messages (i.e. tolerable time for no incoming messages), without any other delay method (i.e., sleeping the thread or using .ready() to check before reading), a SocketTimeoutException will be thrown to stop the blocking of an potentially tampered connection.
- Moreover, without fail-stop assumption within this P2P distributed system, it is important to use such timeout value set tolerance on when to notify the other transmitting party that the remote peer/ server process has deemed as failed. Providing such threshold allow other parties to systematically terminate connections that could lead to future errors (e.g., a hanging peer suddenly sending through blockReplys).

→Theoretically, this timeout to is to catch failures like:

1. Exceptions in processing the requests from peers or the client failed to process or execute the file information received and no longer can generate generate the corresponding responses as an exception is thrown.
2. Process termination, where the termination of a connected peer or an index server will result in no data transmission for the other party to read and resulting in timeout value to elapse and capture such ungraceful termination. (Moreover, this also handled on the peer side that any EchoDownloder and Echo Server thread are termination gracefully if they were ever interrupted by the main Peer process and sending Goodbye messages accordingly).
3. OS/hardware and power supply failure (e.g., storage device or network device) failure similarly can be captured if any remote side of hardware has crashed
4. Network failure can be captured as significant failure/delay of routing will be captured by the socket time out and allow both sides to terminate the ineffective sharing channel.

→The most obvious failure in theory that cannot be captured in this timeout value, where any failed method calling or unexpected logical false resulted in returning deviated messages (e.g., empty sharing IndexElement or non-sharing status).

- In such situations, the peer or the index itself has to handle such errors in order for the correct decisions to be made on the next message. For example, We are expecting that the hitting elements are from sharers that are actually sharing such file, but if we received non-sharing status, such deviated indicator is still valid from the perspective of receiving message on time, but it does not detect such error.

→The timeout value cannot be too small, which considering a fixed timeout option of 1000ms (1 second blocking time) may reduce the effectiveness of communication. Especially for the P2P downloading procedures, where messages may contain large processing time or uploading / downloading time for block requests, it is inappropriate to set such low tolerant and closing a socket that is actually communication. Vice versa, we will not be able to catch ineffective communication channels if our timeout value is too long, and losing our advantage of capturing the listed failures above.

→In the sense of sharing files, an appropriate value of timeout should be calculated basing on the size of block the sharer tends to use (described in its fileDescr) with account to the average uploading (i.e., peer sharer side) and downloading (i.e., peer downloader side) latency. Applying graph theory, let $\boldsymbol{\tau_{sharer,Peer}}$ $\boldsymbol{\tau_{Peer,sharer}}$be the latency for passing such block size of data, then an optimal timeout value for such socket to awaiting for Message transmission

$$\frac{\left(\sum_{sharer\ i\ ,peer\ j} \boldsymbol{\tau_{sharer,Peer}} + \sum_{sharer\ i\ ,peer\ j} \boldsymbol{\tau_{sharer,Peer}}\right)}{\sum_{sharer\ i\ ,peer\ j} \mathbf{1}}/2$$

- Note, this approach may not be feasible as the network condition and block partition of peers is dynamic and confirmating of such averaged value is hard
- Alternatively, we could implement an approximation at each peer based on their own pairs' network conditions and partition strategies in their fileDescr.

Question 4:

→Since the server manage only the index of sharing files and sharers, each request and response the server makes can be assumed as constant (i.e., each message sending and receiving by the index serve is constant).

$$\boxed{P_{i,\text{network}} = \sum_{(i,i')\in\mathcal{E}_n} \omega_{i,i'} + \sum_{(i',i)\in\mathcal{E}_n} \omega_{i',i}}$$

- For each handshake, index server will nee age being received, 2 messages being send
- For each Request, index server will process **2** messages, with 1 request message being received, 1 reply message being send.

1. From the perspective of resources requirements and machine capacity is required for the Index server process, where each requesting peer may take up heterogeneous capacities depending on the amount of files they tend to download and share.

→Then, let $P_{j,\text{network}}$ be the network capacity that peer j required and n(j) be the uploading/downloading bit-rates of each message, the total network resources is contingent to the number of files to be shared and downloaded.

- Number of shareQuest will be the request on completed files and those downloading files that is concurrently being shared (Assuming the sharer's secret is never updated (shareRecord is only called when sharing, not for updates)
- Number of LookUpRequest is simply the number of files to be downloaded
- Number of SearchRequest is a bit more complex provided a search can enable many hits in the table for peer to download, we will assume each search up is for each download.
- Number of dropShareRequest is assumed to be only when the peer is existing the distributed system, which will be the number of shared and downloading files.

$P_{j,\text{network}} = resources\ to\ reply\ shareRequest + resources\ to\ reply\ lookUp + resources\ to\ replySearchRequest + \ resources\ to\ reply\ dropRequest$

$P_{j,\text{network}} = (No.seed + No.download) \times (\mathbf{3} + \mathbf{2})n(j) + No.download \times (\mathbf{3} + \mathbf{2})n(j) + No.download \times (\mathbf{3} + \mathbf{2})n(j) + (No.seed + No.download) \times (\mathbf{3} + \mathbf{2})n(j)$

$P_{j,\text{network}} = (2 \times No.seed + 4 \times No.download) \times (\mathbf{3} + \mathbf{2})n(j)$

- Noticeably, due Server's single request-per-connection setting, Network usage are proportionally occupied by redundant handshaking procedures.

$$C_{network} = \sum_{j=1}^{n} P_{j,network}\ , for\ n\ peers\ such\ that\ C_{network} < NeCTAR_{network}$$

→Let the storage space of one sharing IndexElement to be fixed as S(ie), and $P_{j,\text{Storage}}$ to be the storage spaced needed in <u>IndexMgr's md5Map</u> and <u>sharerMap</u> for peer j's record, the storage requirement can be modelled as follow:

- $P_{j,\text{Storage}} = 2 \times (No.seed + No.download)$
- Which is highly contingent to the number files each peer tends to share and download (se the argument under Queuing theory).

$$C_{Storage} = \sum_{j=1}^{n} P_{j,storage}\,, for\ n\ peers\ such\ that\ C_{Storage} < NeCTAR_{storage}$$

2. From the perspective of Request processing, the computation time can be the main limit in the measurement of index server scalability, in which the IO Thread facilitates the incoming connections through an operating system queue (i.e., LinkedBlockingDeque in this case), where the index server needs to process single request fast enough for it maintain the sufficient idle queuing capacity instead of dropping.

→According to the Queuing Theory: we can estimate the probability of socket dropping rate as a measurement of to how many peers can this index server support:

- $Drop\ probability\ = \frac{\lambda(mean\ rate\ of\ incoming\ requests)}{\mu(mean\ rate\ of\ serving\ connections\ by\ IndexMgr)}$
- Note, these estimated expectation value can be modelled through exponential distributions to assist the prediction of index server's scalability.

→Since no code can inserted into Server package, we cannot really verify this model, but ideally within a time period, we could have some ArrayList or HashMap to record the time taken for the IO Thread to receive another socket and the time taken for the Server (calling IndexMgr) to respond to requests. Then we can estimate both λ and μ using exponential distribution.

- Then, the occupancy (i.e., effectiveness) of the socket incoming rate can be expressed as $\lambda_{effective} = \lambda(1 - drop\ probability)$, where we can approximate the scaling limit of such index server.
- That is once the occupancy rate (fraction of processing time) approaches the index server's expected request processing rate($\lambda_{eff} \geq$ μ), the IOThread will eventually become 100% occupied due to the insufficient processing rate for such incoming rate.

→Assuming each peer is at least seeding a complete and unique file (i.e., the root of this file's P2P sharing), and every other peer is to download at least one file from each other, then the number of LookUPRequest and concurrent shareRequest grows at an exponential rate (n(n-1)).

→Aside from the increasing unique file sharing, we cannot assume the processing time of each request to be constant provided the sharing record will grow exponentially too (i.e., each peer will share its own copy of Index element for n-1 files). Therefore, processing time with looping processes or existence checking methods (e.g., LookUp and Search method of indexMgr involves loop through all the Index Elements to check hits on file hashes and keywords accordingly) will grow on a scalable level too, which further burdens decrease the expected processing rate μ.