**System Description**

**Protocol requirements:**

Use Java to write the system. To achieve the Peer to peer file sharing system, we need a client program that can run multiple processes simultaneously and a server program. So the server program need to generate a server socket object to receive the request from multiple clients. And each clients have to generate a socket object to communicate with server and also a server socket object to receive chunk downloading requests from other peers. The port number stored in the server’s chunksPossessList property and peerItemArrayList property should be the clients’ server socket port(used to communicate transmit chunks to other peers) rather than socket port number(which is used to communicate with the server). So that, when a peer get the reply of chunk locations, they can send the request to the target peer directly.

**Basic Requirements:**

To meet the requirement of multiple connections, in both client and peer sides, we should constantly listen for socket connection request from multiple peers. Once we accept a connection request, we store it in a request queue, and there is another listener peeking the first element of the queue using while loop. Once there is a request peeked by the listener, we create a new thread to generate new socket, output stream, input stream to process it.

In order to have the function of parallel downloading, when we get the chunk location reply from the server, we generate a number of threads according to the number of chunks of the target file. Each thread will send a connection and file chunk request to a random peer shown in the peerPortList.

Once the peer has received the whole chunk of file, which also means receive the reply from the other peer and test the reply is exactly what it wants, function chunkRegister() will be call to send a request to server. The request contains the file name, chunk number, peer’s server socket port. When server receive the Json message, it will change the chunksPossessList property. Then, when another peer request the chunk location, server will send back the newly update locations(peer ports).

To make sure there is no conflict of modifying the same variable at the same time. We need to use synchronized() carefully. And In order to make sure the data integrity, we need to use MessageDigest class which is provided by java to calculate the hash value of each chunks and store it in the chunksPossessList. If the peer get the chunk that has different hash value that it get from the server, it means the chunk has been changed by others. So it must request it again from other peers.

**Main Class Structure**

Class Server

|  |  |
| --- | --- |
| **Attributes** | **data type** |
| peerItemArrayList | ArrayList<PeerItem> |
| sharedFileList | ArrayList<String> |
| chunksPossessList | ArrayList<FileChunks> |
| requestQueue | RequestQueue |

Class PeerItem

|  |  |
| --- | --- |
| **Attributes** | **data type** |
| peerID | int |
| peerIP | String (always “127.0.0.1”) |
| peerPort | int |

Class client

|  |  |
| --- | --- |
| **Attributes** | **data type** |
| localServerPort | int |
| localFiles | ArrayList<FileItem> |
| chunkReceive | int |

Class FileChunks

|  |  |
| --- | --- |
| **Attributes** | **data type** |
| containFileName | String |
| chunkSeqNum | int |
| peerPortList | ArrayList<Integer> |
| hashValue | String |

Class RequestQueue

|  |  |
| --- | --- |
| **Attributes** | **data type** |
| requestList | LinkedList<Socket> |
| requestContents | LinkedList<JSONObject> |

**Communication interface document**

We use Json format to communicate between server and clients, below is Json format of each interfaces.

**Peer register：**

Request:

{

"requestType": 0，

"fileNumber": ""，

"localServerPort": ""，

"files": [

{

"fileName" : "",

"fileLength" : ""

}

]

}

E.g.

{"requestType": 0,"fileNumber":2,"files":[{"fileName":"li","fileLength":12},{"fileName":"jing","fileLength":10}]}

Reply:

{ "status": "success" }

**File list request:**

request:

{"requestType": 1}

Reply:

{"fileNames": ["<fileName>", ...]}

**File Locations Request:**

Request:

{

"requestType": 2,

"requestFileName": "",

}

Reply:

{

“chunkLocation”: [

{

"chunkFile": "<filename>",

"chunkSeqNum": "2",

"peerPortList": []

},{

"chunkFile": "<filename>",

"chunkSeqNum": "2",

"peerPortList": []

}, ... ...]

}

E.g. {"chunkLocation":[{"peerPortList":[49274],"chunkSeqNum":0,"chunkFile":"file1"},{"peerPortList":[49274],"chunkSeqNum":1,"chunkFile":"file1"}]}

**File Chunk Request:**

Request:

{

"fileName": "",

"chunkSeqNum": "",

}

E.g.

{"fileName": "file1","chunkSeqNum": 1}

Reply:

{

"fileName": "",

"chunkSeqNum": "",

}

**Chunk Register Request:**

Request:

{

"requestType": 4,

"portNumber": "",

"fileName": "",

"chunkSeqNum": "",

}

E.g.

{"fileName":"file1","requestType":4,"chunkSeqNum":0,"portNumber":9092}

Reply:

{ "status": "chunk register success" }

**Close connection:**

{

"requestType": 5,

"localServerPort": ""

}

Use field name “requestType” to represent different request. Correspondence between request types and values are as follow:

|  |  |
| --- | --- |
| **requestType** | **Seq number** |
| Peer register | 0 |
| File list request | 1 |
| File Locations Request | 2 |
| File Chunk Request | 3 (not necessary because only one kind of request between peer and peer) |
| Chunk Register Request | 4 |
| Close Connection | 5 |