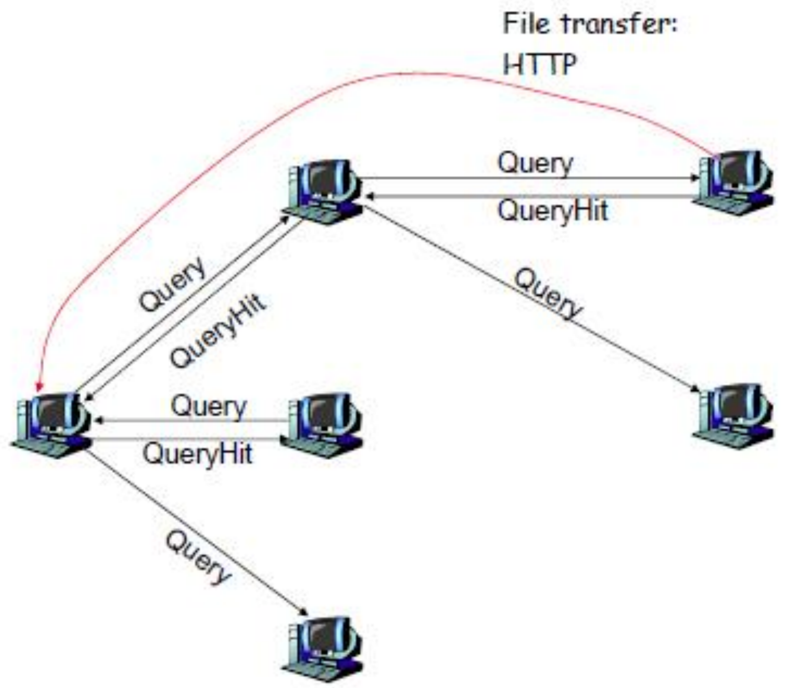
5. Design document

Architecture Overview

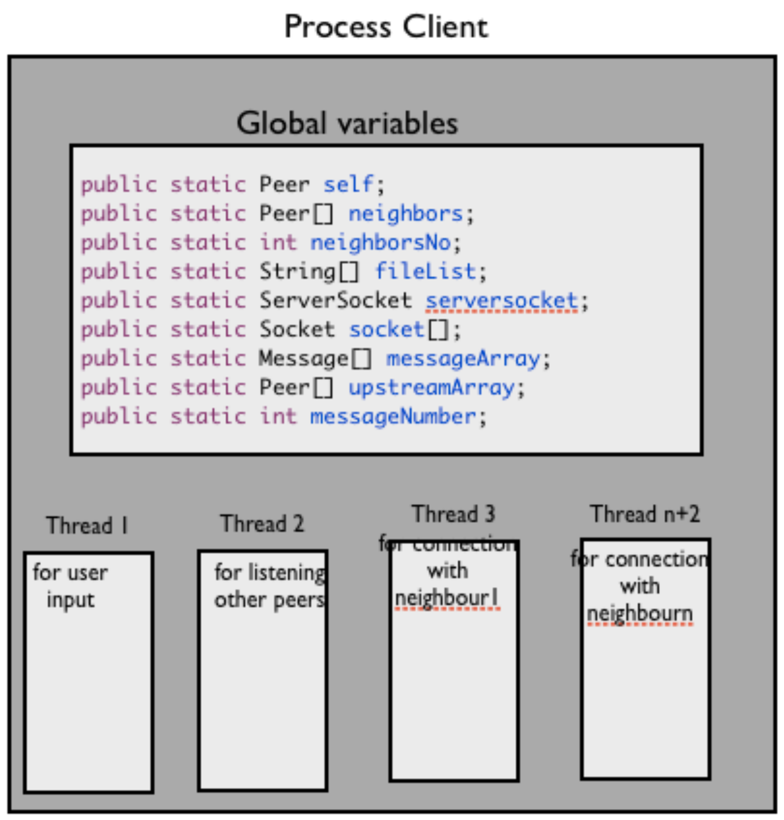
The system is designed as a typical **Gnutella network** architecture, which implementing peer-to-peer delivering. The unique component in this system is Peer. In the peer-to-peer module, Peer functions as a client and also as a server. In this module, one Peer is as the source sending data and the other one works as destination accepting data, and vice versa. Once a Peer sends out a message, the message can be spread through the whole network.



The figure showed above is a typical **Gnutella network** P2P architecture.

Client Design

Different from programing assignment 1, in this assignment, there is a server in each peer instead of a centralized one. For example, suppose if a peer has n neighbors, we purpose there are **n+2** threads. N threads for connection with n neighbors, 1 thread for serversocket to listening other peers, 1 main thread for user input.


Process Client
Global variables
Thread 1
for user input
Thread 2
for listening other peers
Thread 3
for connection with neighbour1
Thread n+2
for connection with neighbourn

```
public static Peer self;
public static Peer[] neighbors;
public static int neighborsNo;
public static String[] fileList;
public static ServerSocket serversocket;
public static Socket socket[];
public static Message[] messageArray;
public static Peer[] upstreamArray;
public static int messageNumber;
```

Client Architecture

The figure above is the structure of the client architecture. It has at most n+2 threads. When one peer has created, we need to establish the connections with other clients. We hope to store the connections instead of connecting to others when a communication is needed. This design can improve performance by decreasing connection establishing overhead. We can save a lot of time when we handle 1000 or more connections by avoiding too much overhead with our design.

When the client is created, we store all the sockets, each of which connects to one of the client' neighbors. These connections should be global in the process so that each thread could get these variables. To achieve this feature in java, we use **static** variable.

Because in our design, 3 different kinds of threads are required, we need to implement creating multiple threads in one process. 3 threads are: **1.**

User input thread; 2. Listening thread; 3. Neighbour connection thread. We name them thread A, thread B and thread C respectively for convenience.

Thread A is from the client's main function. Thread B and thread C we need to implement. We propose two classes, **class Listener** and **class Handler**, each implement the runnable interface and can fork one kind of thread.

Connections Initialization

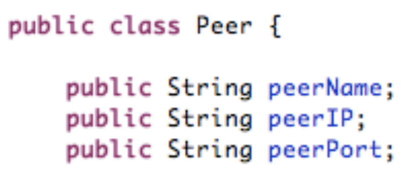
At first, we propose that when a client is created, the connections with its neighbors are automatically initialized. But it fails, the reason is that the real network is statically loaded from the configuration file, each client reads its neighbors from the configuration file. We cannot ensure that all the clients in the network are created simultaneously, thus some connections may fail, which results in the network doesn't work correctly.

Our final design is : when a client is created, its connections with its neighbors are not initialized. When a user inputs a command indexed with 1, it starts to create the connections.

Message Structure

A good message structure is needed, we spend much time on discussing the message data structure.

A peer has its own peer ID, the data structure for this is :

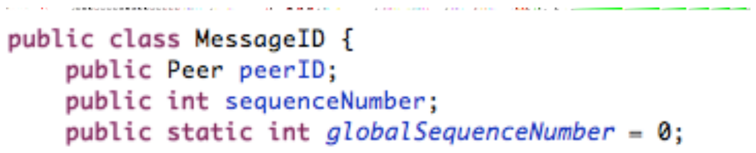


```
public class Peer {

    public String peerName;
    public String peerIP;
    public String peerPort;
```

The peerName is a human-friendly string, such as p1, p2, p3.

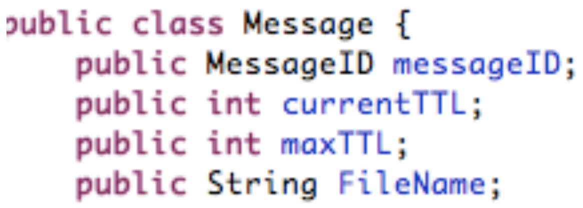
Each message also need a messageID.



```
public class MessageID {
    public Peer peerID;
    public int sequenceNumber;
    public static int globalSequenceNumber = 0;
```

SequenceNumber starts from 0. When a new message is created, the sequenceNumber increments by 1. We implement this by using a static variable.

Then Message class is as follows.

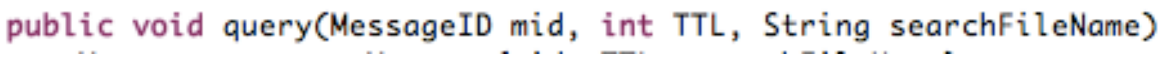


```
public class Message {
    public MessageID messageID;
    public int currentTTL;
    public int maxTTL;
    public String FileName;
```

The currentTTL is decremented by 1 when it received by a client. We incapsulate the filename into the message.

Query Implement

The interface of the query function is :



```
public void query(MessageID mid, int TTL, String searchFileName)
```

The process of implementing query is:

1. When a user choose to search a file, he is required to input the filename and ttl(because the network is static, only the user knows the length of each path).

2. The message sender creates a message and uses a for loop to send a command indexed with 2 to tell the target a message will achieved, and then send the message to its neighbours.

3. When its neighbor gets this message, the message decrements the currentTTL by 1. Suppose the neighbor's name is pA, pA checks if it has the message already. If no, pA stores the message and its upstream client into the messageArray and upstreamArray respectively.

4. Then pA checks if the message's current TTL is 0. If not, forward this message to all its neighbors except the upstream client.

HitQuery implement

When a client, let's say A, gets a message m, it needs to send hitquery message to m' sender. The way we implement this is:

1. A checks if it has the search file, using a boolean variable to store the result, and incapsulate the result into a hitmessage.

2. A checks m's upstream client in the upstreamArray, and send the hitmessage to A's upstream client, B.

3. The upstream B gets the hitmessage, and do the step 2. B checks if it is the message m's sender, if not, sends it to the corresponding upstream client. If B is the sender, print the search result.

4. Repeat the step 2 and 3 until we reach the message's sender.

Obtain implement

Obtain is relatively easy, the user input the download file, download client name, and the current client choose the corresponding socket associated with the download client and obtain the file. It is quite similar with the same implementation in the previous assignment.

Utilization of techniques

This project was implemented by using Java language and the IDE is eclipse. Socket is used to establish the connection between two nodes (server or peer) due to its convenience by hiding the complexity of underlying network and Java's strongly support.

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson in this project is used to transform specific Message to the data structure that can be transferred by Socket. At the receiving terminal, again, Gson retrieves the data from Socket and turns it back to Message that we need.

This project uses the library of Java SE 1.6(JRE System Library) and Google-Gson. The source folder mainly consists of three java files: Client, Listener,

Client: The class of Client is the representation of Peer, mainly responsible for sending request to other peers. It also firstly creates the connection with its neighbors, and generates a Listener thread when it is being initialized.

Listener: This thread is used for creating ServerSocket to accept connection request from other Peer. Once connection accepted, a new thread called Handler is used to handle afterward operation related to that socket connection.

Handler: Handler thread is the class responsible for accepting any request such as Query or Obtain and address it. A special functionality in Handler is that it can forward message from upstream nodes to its neighbors. Meanwhile, those upstream nodes were recorded locally to provide reverse path for re-propagating hit-query message. If a peer has the file that the Query message is looking for, the handler delivers a hit-query message back to its upstream node. When a hit-query message arrives, Handler also forwards it back to the its upstream Peer.

Improvements and Extensions

When the number of Peer grows dramatically, a node may receive large number of requests, which means generating the same number of threads that definitely will bring overheads and harm the efficiency. Introducing threads pool a viable solution. If a Peer needs to send several files to the same destination, it is wasteful to allocate these files into different thread. We can consider batch addressing as a solution