How we did

At the beginning, we learned Gnutella architecture including the function of different messages and Gnutella protocol structure including header and payload. We figure out how Gnutella works and implementation detail.
Then, We look into some source packages like phex, which gives us a top-down architecture view of servant communication.
Next, we design our class structure and implement functions one by one. Specifically, we start from defining fields in messages, then implemented logical socket data transfer as a server or a client. We add the needed classes during development. Besides, we searched on line to gain clues about implementing functions as well as fixing bugs.

Basic skeleton building

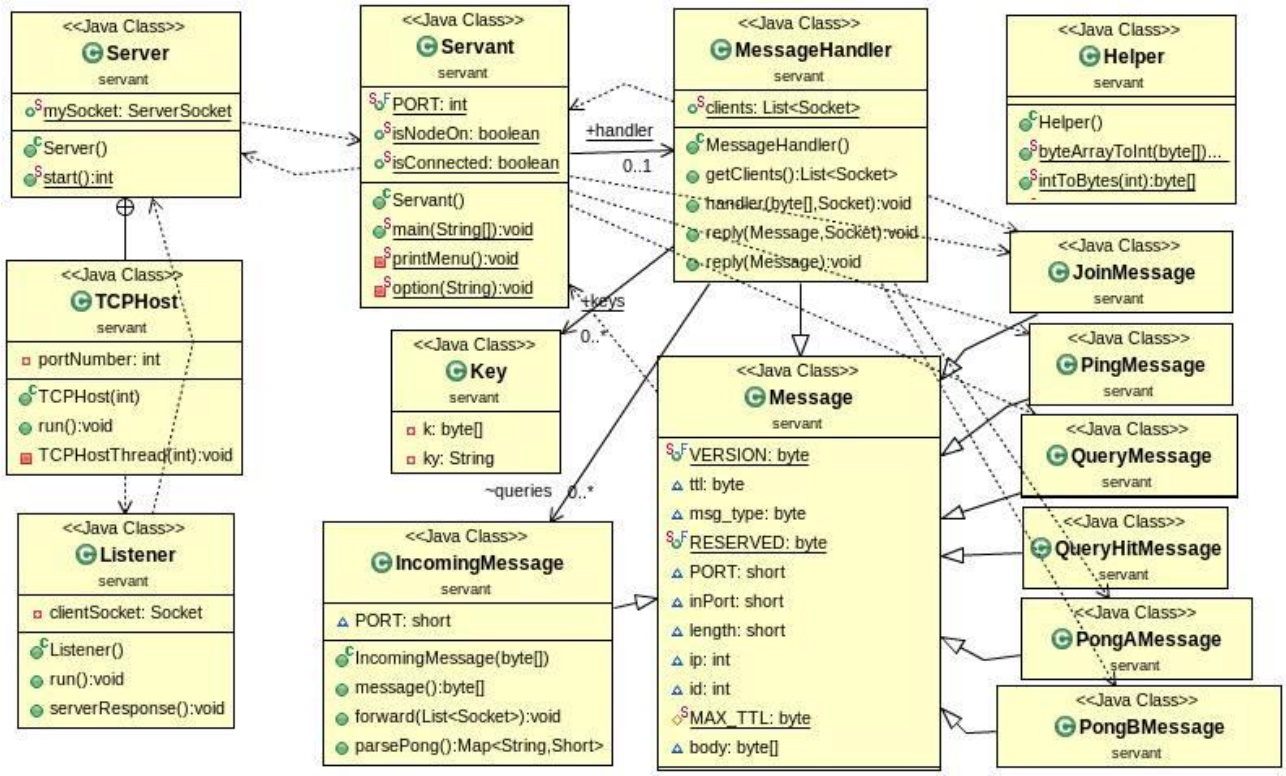
- *Create client/server classes*
  The application is based on client-server model and it serves as both client ans server in the network. We create socket to make connection to other machines. We used the methods getInputStream() and getOutputStream() to produce the corresponding InputStream and OutputStream objects from each Socket. In client class, we define several system input cases through which different types of message sent. In server class, we create a sever socket used to listen for incoming connections.

- *Create listener class and message handler class*

Listener is a runnable target in which server socket capture(accept) a incoming client socket. The listener thread will start after server socket created. Once the connection is build (client socket accepted) is and message handler will be called to handle incoming message.
The message handler class handle all incoming messages based on attributes in message header then redirect to corresponding methods to response.

- *Create messages classes*
  1. Create a base class for all types of messages in which protocol header is defined and corresponding methods used to set/get attributes in header is implemented.
  2. Create all types of message classes which extend the base class. For messages which have body, we utilize an external library to combine its header and body.
  3. Apart from messages sent, we create a class to store and forward incoming message.
  4. In order to set/get the attribute value in the protocol header and body, we create a helper class which fulfill conversion between byte array and other data types.

Java class diagram



Explanation

- *Servant class*
  The main function is implemented and several cases are defined to initiative send messages respectively according to system input.

- *Server class*
  In which the server starts TCPHost thread to listen incoming connection requests.

- *TCPHost class*
  A runnable target inside server class in which server socket is defined at defined port and Listener thread is started.

- *Listener class*
  A runnable target in which the action that server socket accepts the client socket and build connection is implemented.

- *Message class*
  A base class for all types of messages in which protocol header is defined and corresponding methods used to set/get attributes in header is implemented.

- *Message handler class*
  Handle all incoming messages then redirect to corresponding methods to response as well as generate out messages. Therefore, two methods – handle and reply are implemented. Handle method deals with and response incoming messages and reply method initiative sends out messages such as join and ping.

- *Incoming Message class*

A class which stores incoming messages and in message handler class an instance of incoming message is created to response based on the attribute value of incoming message. And for special purpose to expand network or handle Type B Pong message, a method parsePong() is created. In parsePong(), a list (a Hashmap instance) is created in which the ip addr and port of entries from incoming Pong B message viewed as a set of key-value pairs. Therefore, we gain the node information in the network from incoming Pong B messages and then join messages sent.

- *Key class*
  Define key and the function to get/set key. The key will be used in message handler class to handle query message.

- *Helper class*
  In which conversion between byte array and other data types is fulfilled in order to set/get the field value in the protocol header and body.

- *Join Message class*
  Extends from Message class and defines Join message.

- *Ping Message class*
  Extends from Message class and defines Ping A and Ping B message.

- *Query Message class*
  Extends from Message class and defines Query message.

- *Query Hit Message class*
  Extends from Message class and defines Query Hit message.

- *PongA Message class*
  Extends from Message class and defines Pong A message.

- *PongB Message class*
  Extends from Message class and defines Pong B message.

Detail key functions implementation

- *Send Join Request message in order to join the network*

The function implemented as a case in Servant class. The ip addr and port is set through system input (command line). The reply method in message handler is called to send Join Request. And in message handler class, after receiving join response message, the clientsocket is added to the connected client socket list.

- *Handle Join Request message*

The response method which take in two parameters including message id and send Join Response message (with 2 bytes body) is implemented in Join Message class. In message handler class, after receiving join request message, the response method will be called to send reply to the client socket.

- *Send Query message with a search key and handle Query Hit messages*
  The search key is set through system input (command line). A query message class is

created and the function is implemented as a case in Servant class. Query Hit messages is handled in Message Handler class.

- *Publish a key/value pair in the network and respond to a Query message when its search key equals your published key*
  The publish key is set through system input (command line) and it will be added to a key list. Once the Query message is received, firstly check whether is existed, then add the query to the query list. Then check whether the key in the query body matches local key (Traversal key list). If matched, send a query hit, otherwise check the TTL and determine whether forward the incoming messages to other nodes.

- *Join multiple peers to expand network*

We create a list (a Hashmap instance) in which the ip addr and port of entries from incoming Pong B message viewed as a set of key-value pairs. We gain the node information in the network from incoming Pong B messages. We examine and remove duplicate clients in the list and send join messages to other nodes.

- *Send Type B Ping message and handle Type B Pong message to find more node information in the network*
  Create sendPingB method in which ping B message instance created and utilize OutputStream object to send message header (byte array) to one connected socket. Handle Type B pong message part is addressed in expand network function.

- *Respond to Type B Ping message with a Type B Pong message*

Define an incoming message instance through which incoming message id and ttl are gained. Then four parameters including id, ttl, connected client socket list, client socket is taken in to build a Type B Pong message. Finally, call reply method to send Type B Pong message.

- *Forward Query messages with loop avoidance to help other node to find a resource*
  Once the Query message is received, firstly check whether is existed, then add the query to the query list. Then check whether the key in the query body matches local key (Traversal key list). If matched, send a query hit, otherwise check the TTL and determine whether forward the incoming messages to other nodes.

- *Send Type A Ping message and handle Type A Pong message to check if your neighbours are alive*
  Similar (and easy) to Type B

- *Respond to Type A Ping message with a Type A Pong message. Tell your neighbour that you are still alive*

Similar (and easy) to Type B

Referencing jars

commons-lang3-3.3.2.jar
commons-lang3-3.3.2-javadoc.jar
guava-15.0.jar