Computer Networks PA #2 디자인 및 구현 문서

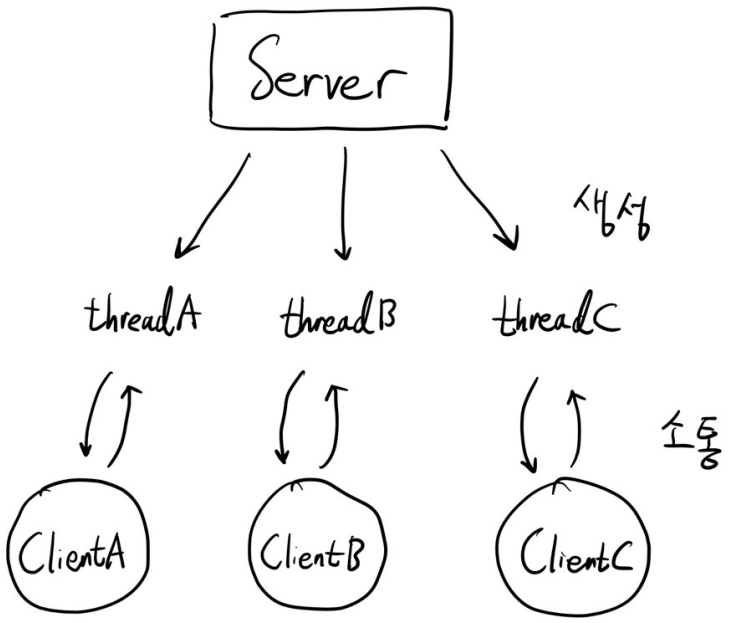
2020059152 오주영

1. 프로그램의 설계 및 구조의 기술

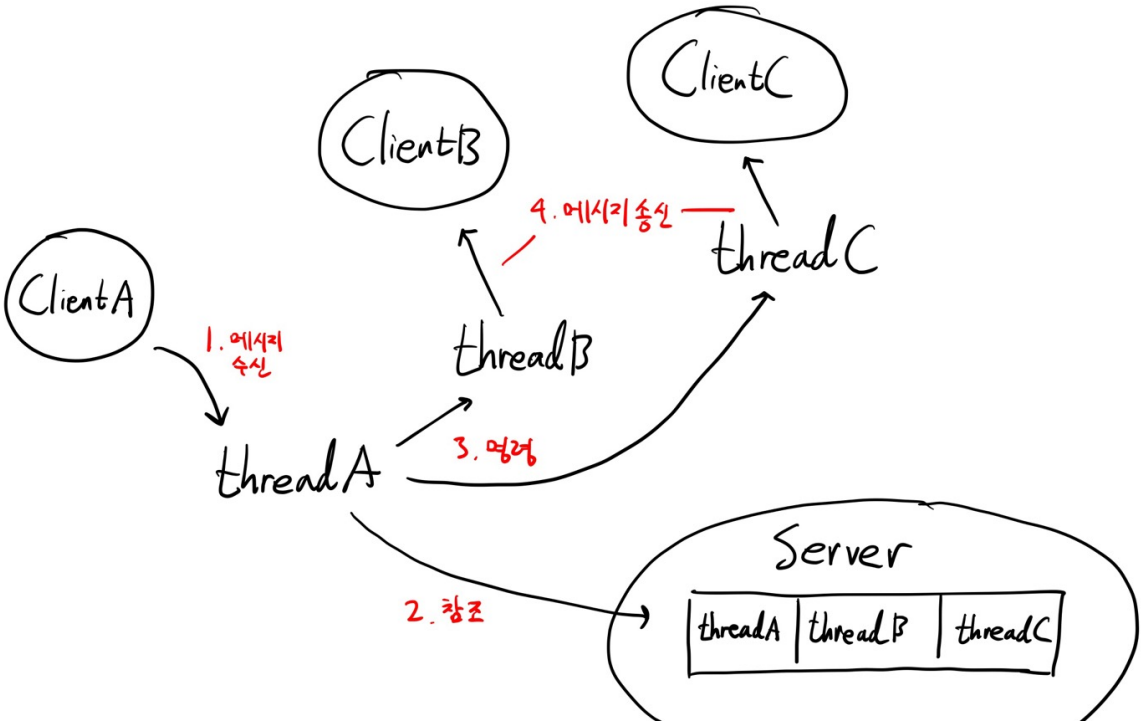
TCP를 이용한 클라이언트 - 서버 방식의 오픈 채팅 프로그램에 대한 구현 문서이다. 구현한 클래스들과 그 메서드들에 대한 설명에 앞서, 전체적인 설계 구조에 대해 설명하겠다.

프로그램의 설계 및 구조

TCP 연결은 UDP와 달리 point to point 프로토콜이기에, 기본적으로 unicast이 원칙이다. 서버 하나가 동시에 여러 클라이언트와 소통하기 위해서는, 각 클라이언트에 대해 TCP 연결을 가져야 하는 것이다. 또, 서버가 두개 이상의 연결을 동시적으로 처리해야하므로, 각 연결은 쓰레드처럼 작동할 수 있어야 한다. 따라서 서버는 새로운 클라이언트의 연결 요청을 받을 때마다 그 클라이언트에 대한 연결 쓰레드를 생성해야 한다. 즉, 서버는 연결 쓰레드를 생성할 뿐이고 실제 클라이언트와의 소통은 이 연결 쓰레드가 담당하는 것이다.



위와 같은 디자인을 고려해보자. 서버 하나가 세 클라이언트에 대한 각각의 연결 쓰레드를 생성한다. 각 클라이언트는 각자의 연결 쓰레드와 소통을 할 수 있지만, 클라이언트 서로간의 소통은 불가능하다. 각 연결 쓰레드는, 앞서 말했듯 point to point 방식이기 때문이다. 예를 들어 위 그림에서 threadA는 오로지 ClientA이랑만 소통할 수 있고, threadB와 threadC의 존재조차 모른다. 즉 클라이언트 간의 소통이 이루어지기 위해서는 한 연결 쓰레드는 다른 연결 쓰레드들의 정보를 알 수 있어야 한다.



이를 위해 위와 같은 디자인을 채택했다. 우선 서버는 모든 연결 쓰레드에 대한 리스트를 저장해놓는다. 이후 각 연결 쓰레드는, 클라이언트로부터 메시지를 수신할 때마다 이 리스트를 참조하고, 다른 연결 쓰레드들에게 각자의 클라이언트에게 메시지를 송신하라는 명령을 전달한다. 위 예시는 하나의 채팅방에 대한 예시이고, 채팅방이 여럿 있을 경우엔 서버에 여러 리스트를 저장해두면 된다. 큰 틀은 이러하고, 이제 구현한 클래스와 메서드들에 대해 기술하겠다.

Server 클래스

멤버 변수

```
private ServerSocket chatSocket;
private ServerSocket fileSocket;
private HashMap<String, ArrayList<ConnectionThread>> chatrooms;
private HashMap<String, byte[]> files;
```

이 프로그램에서 각 클라이언트는 두 포트번호를 이용해 하나의 메시지 송수신 연결과, 하나의 파일 송수신 연결을 수립한다. 따라서 Server 측엔 이 두 연결 요청을 기다릴 welcoming socket이 각각 필요하고, chatSocket과 fileSocket이 이 역할을 한다. chatrooms는 HashMap이며, 특정 채팅방 이름에 대응되는 연결들(이후 설명할 ConnectionThread)에 대한 정보를 저장한다. files 역시 HashMap으로, 특정 파일명에 대응되는 파일의 바이트 정보를 저장한다.

생성자

```
public Server(int port1, int port2) {
    chatrooms = new HashMap<>();
    files = new HashMap<>();
    try {
      chatSocket = new ServerSocket(port1);
      fileSocket = new ServerSocket(port2);
    } catch (Exception e) { e.printStackTrace(); }
}
```

Server는 두 포트번호를 인자로 받아 chatSocket과 fileSocket을 각각에 바인딩해준다.

acceptClients 메서드

```
public void acceptClients() {
  try {
    while (true) {
      Socket clientChatSocket = chatSocket.accept();
      Socket clientFileSocket = fileSocket.accept();

      new ConnectionThread(clientChatSocket, clientFileSocket).start();
    }
  } catch (Exception e) { e.printStackTrace(); }
}
```

Server는 Always-On 방식으로, 항시 클라이언트의 연결 요청을 기다린다. 새 연결 요청을 받으면 이를 처리할 연결 쓰레드ConnectionThread 객체를 생성해주고 이 쓰레드를 시작시켜준다.

main 메서드

```
public static void main(String[] args) {

  int port1 = Integer.parseInt(args[0]);
  int port2 = Integer.parseInt(args[1]);

  new Server(port1, port2).acceptClients();

}
```

실행인자로 포트번호를 둘 받아와 Server에 바인딩해주고 acceptClients 메서드를 호출해 클라이언트의 연결을 기다린다.

여기까지가 Server 클래스의 전부이다. 위 언급했듯 Server는 ConnectionThread를 생성시킬 뿐이고, 실제 클라이언트와의 소통은 ConnectionThread가 담당한다. 이제 ConnectionThread에 대한 설명을 시작하겠다.

ConnectionThread 클래스

들어가기 앞서 언급할 사항이 몇가지 있다: ConnectionThread는 Thread 역할을 하기 위해 Thread를 상속한다. 또 ConnectionThread는 Server의 helper 클래스로만 쓰이고, 여러 ConnectionThread는 한 Server 객체의 chatrooms과 files 변수에 접근할 수 있어야 하기 때문에 Server의 Inner 클래스로 구현하였다.

멤버 변수

```
private ArrayList<ConnectionThread> members;
private BufferedReader reader;
private PrintWriter writer;
private DataInputStream dataInputStream;
private DataOutputStream dataOutputStream;
private String roomName;
private String username;
```

members는 같은 채팅방에 속한 ConnectionThread들에 대한 리스트이다. 즉 클라이언트가 처음 접속할 당시에는 null값을 가지고, #CREATE이나 #JOIN 명령어를 던지면 그 해당 채팅방 멤버들로 갱신이 된다. 참고로, 실제 ConnectionThread들에 대한 정보는 Server 클래스에서 가지고 있다는 걸 명심해야 한다. 즉 members는 리스트라기보다는, 그런 리스트에 대한 포인터로 이해하면 된다. reader와 writer는 각각 클라이언트와 메시지를 수신, 송신할 때 사용된다. dataInputStream과 dataOutputStream은 각각 클라이언트와 파일을 수신, 송신할 때 사용된다. roomName과 userName은 각각 현재 클라이언트의 속해 있는 채팅방 이름, 그리고 클라이언트의 유저네임을 저장한다.

생성자

```
public ConnectionThread(Socket clientChatSocket, Socket clientFileSocket) {
  try {
    reader = new BufferedReader(new InputStreamReader(clientChatSocket.getInputStream()));
    writer = new PrintWriter(new OutputStreamWriter(clientChatSocket.getOutputStream()));

    dataInputStream = new DataInputStream(clientFileSocket.getInputStream());
    dataOutputStream = new DataOutputStream(clientFileSocket.getOutputStream());
```

```
  } catch (Exception e) { e.printStackTrace(); }
}
```

인자로 받아온 두 소켓의 각 IO Stream을 이용해 멤버변수들을 초기화시킨다.

sendMessageToClient 메서드

```
public void sendMessageToClient(String message) {
  try {
    writer.println(message);
    writer.flush();
  } catch (Exception e) { e.printStackTrace(); }
}
```

이 연결에 대응되는 클라이언트에게 메시지를 전송한다. 위 그림 예시에서 threadA가 ClientA에게 메시지를 보내는 것과 같다.

sendMessageToMembers 메서드

```
public void sendMessageToMembers(String message) {
  for (ConnectionThread member : members)
    if (member != this) member.sendMessageToClient(message);
}
```

현재 같은 채팅방, 즉 members 리스트에 있는 다른 ConnectionThread들의, 위 기술한 sendMessageToClient 메서드를 호출시킨다. 이 메시지를 송신한 클라이언트에게 같은 내용을 송신하지 않도록 비교과정을 거친다. 위 그림 예시에서, ClientA로부터 메시지를 수신한 threadA가 threadB와 threadC에게 명령을 내리는 것과 같다.

joinRoom 메서드

```
public void joinRoom(String newRoomName, String newUsername) {

  ArrayList<ConnectionThread> newRoom = chatrooms.get(newRoomName);

  if (newRoom == null) {
    sendMessageToClient("### CHATROOM <" + newRoomName + "> DOES NOT EXIST");
    return;
  }

  if (members != null) leaveRoom();

  roomName = newRoomName;
```

```
  username = newUsername;

  members = newRoom;
  members.add(this);

  sendMessageToMembers("### " + username + " has joined the room.");
  sendMessageToClient("### Successfully joined room " + roomName);

}
```

클라이언트로부터 #JOIN 명령을 받았을 때 호출되는 메서드이다. 우선 HashMap인 chatrooms에, get 메서드를 통해, 인자로 받아온 채팅방 이름에 대응되는 ConnectionThread 리스트가 존재하는지 확인한다. 존재하지 않을 시 클라이언트에게 그렇다는 메시지를 전송하고 종료한다. 존재한다면, roomName과 username을 갱신해주기 앞서 이미 어떤 방에 참여하고 있는지 확인한다. 그럴 경우 이후 기술할 leaveRoom 메서드를 호출해 먼저 기존 방에서 퇴장한다. 다음 members도 이 리스트로 갱신해주고, 이 리스트에 자신을 추가한다. 마지막으로 다른 멤버들과 이 클라이언트에게 참여 소식을 전송한다.

createRoom 메서드

```
public void createRoom(String newRoomName, String newUsername) {

  if (chatrooms.containsKey(newRoomName)) {
    sendMessageToClient("### CHATROOM <" + newRoomName + "> ALREADY EXISTS");
    return;
  }

  if (members != null) leaveRoom();

  roomName = newRoomName;
  username = newUsername;

  members = new ArrayList<ConnectionThread>();
  members.add(this);

  chatrooms.put(newRoomName, members);

  sendMessageToClient("### Successfully created room " + roomName);

}
```

클라이언트로부터 #CREATE 명령을 받았을 때 호출되는 메서드이다. 위 joinRoom 메서드와 상당히 유사한데, 차이점이 몇몇 있다. 우선 chatrooms에 인자로 받아온 채팅방 이름이 존재하면, 이미 존재하는 채팅방이라는 메시지를 클라이언트에게 전송하고 종료한다. 또 members를 갱신할 때 기존에 없던 새로운 방을 만들어야 하므로, new 키워드를 통해 새로운 리스트를 만든다. 다음 이 새 리스트에 자신을 추가하고, 이 리스트를 Server의 chatrooms에 put 메서드를 통해 추가해 준다. 마지막으로 해당 클라이언트에게 성공적으로 채팅방을 만들었음을 알린다.

leaveRoom 메서드

```
public void leaveRoom() {

  if (members == null) {
    sendMessageToClient("### YOU ARE NOT IN A ROOM");
    return;
  }

  members.remove(this);
  sendMessageToMembers("### " + username + " has left the room.");
  members = null;
  sendMessageToClient("### Successfully left room " + roomName);
}
```

클라이언트가 기존 채팅방에서 퇴장할 때 호출되는 메서드이다. 우선 members가 null인지, 즉 참여하고 있는 채팅방이 없는지 확인한다. 위 joinRoom과 createRoom에서는 이 메서드를 호출하기 전 이 비교를 이미 거치지만, 클라이언트가 #EXIT 명령을 던졌을 때를 위한 과정이다. 만약 참여하고 있는 방이 없다면 그렇다는 메시지를 전송하고 종료한다. 만약 있다면, 즉 members가 null이 아니라면 members에서 자신을 제외한다. 다음 다른 멤버들에게 퇴장 소식을 전하고 members를 null로 갱신해준다. 마지막으로 해당 클라이언트에게 성공적으로 퇴장했음을 알린다.

showStatus 메서드

```
public void showStatus() {

  if (members == null) {
    sendMessageToClient("### YOU ARE NOT IN A ROOM");
    return;
  }

  String status = "\n"
      + "### In Chatroom: <" + roomName + ">\n"
      + "### With <" + members.size() + "> total members\n"
      + "### Joined as <" + username + ">\n";

  for (ConnectionThread member : members) {
    status += "### " + member.username;
    if (member == this) status += " (YOU)";
    status += "\n";
  }

  sendMessageToClient(status);

}
```

클라이언트로부터 #STATUS 명령을 받았을 때 호출되는 메서드이다. 우선 참여하고 있는 채팅방이 있는지 확인한다. 존재한다면 그 채팅방에 대한 정보를 출력하는데, 이 정보에 속하는 내용은 다음과 같다: 채팅방 이름, 총 멤버 수, 나의 유저네임, 참여하고 있는 모든 멤버들의 유저네임.

getFileFromClient 메서드

```
public void getFileFromClient(String fileName) {

  new Thread(new Runnable() {
    @Override
    public void run() {
      try {
        int fileSize = dataInputStream.readInt();
        if (fileSize == -1) return;

        byte[] fileBytes = new byte[fileSize];

        int offset = 0;
        int stride = 64000;

        while (fileSize > 0) {
          int len = stride < fileSize ? stride : fileSize;
          dataInputStream.read(fileBytes, offset, len);
          offset += stride;
          fileSize -= stride;
        }

        files.put(fileName, fileBytes);
      } catch (Exception e) { e.printStackTrace(); }
    }
  }).start();

}
```

클라이언트로부터 #PUT 명령어를 받을 때 호출되는 메서드이다. 클라이언트가 보내는 파일을 저장하는 메서드로, 다른 프로세스들과 충돌하지 않도록 쓰레드를 사용한다. 이 메서드는 이후 설명할 Client 클래스의 putFile 메서드에 대응된다. 위 코드에 대한 이해를 돕기 위해 putFile 메서드에 대한 간략한 설명을 하자면, 우선 Client는 자신이 보낼 파일의 크기를 정수 형태로 전송한다. 하지만 그 파일이 존재하지 않을 시 -1을 전송한다. 따라서 이 getFileFromClient 메서드에선 먼저 Client로부터 받을 파일의 크기 정보, fileSize를 수신하고, 이 값이 -1이라면 Client 측에서 문제가 있었음을 알고 메서드를 종료한다. 그 외 -1이 아닌 fileSize를 받으면, 이 크기의 바이트 배열 fileBytes를 생성해 파일을 수신할 준비를 한다. 또 putFile 메서드에선, 송신하는 64000 바이트마다 진척도를 표기하기 위해 64000 바이트 단위로 파일을 송신한다. 때문에 getFileFromClient 메서드 측에서도 64000 바이트 단위로 수신해야 한다. 위 사용한 DataOutputStream의 read 메서드는 3개의 인자를 사용한다. 첫째는 바이트 배열로, 수신하는 내용을 저장할 위치를 정한다. 둘째는 offset 인덱스인데, 내용을 저장할 배열의 어떤 인덱스부터에 저장할 것인지를 정한다. 셋째는 길이로, 읽어올 바이트의 수를 정한다. 예를 들어 dataInputStream.read(b, 4, 5)는 스트림으로부터 바이트 5개를 읽어온 뒤 이를 b의 4번째 인덱스부터 8번째 인덱스에 저장한다. 따라서 64000 바이트 단위로 수신하기 위해, offset을 증가시켜주며 내용을 읽어오도록 한다. 이 반복문에서 fileSize는 남은(아직 수신하지 않은) 바이트의 수를 뜻하고, len은 매 반복마다 몇 바이트를 읽어올지에 대한 변수이다. 전체 수신할 바이트의 수가 64000보다 적을 때 64000 바이트를 읽으려 하지

않도록 제어한다. 이렇게 모든 바이트를 fileBytes에 저장한 다음, Server 클래스의 files HashMap에 그 파일명과 짝지어 추가해준다.

pushFileToClient 메서드

```
public void pushFileToClient(String fileName) {


  new Thread(new Runnable() {
    @Override
    public void run() {

      byte[] fileBytes = files.get(fileName);

      try {
        if (fileBytes == null) {
          dataOutputStream.writeInt(-1);
          sendMessageToClient("### FILE <" + fileName + "> DOES NOT EXIST");
          return;
        }

        int fileSize = fileBytes.length;
        dataOutputStream.writeInt(fileSize);

        int offset = 0;
        int stride = 64000;

        while (fileSize > 0) {
          int len = stride < fileSize ? stride : fileSize;
          dataOutputStream.write(fileBytes, offset, len);
          offset += stride;
          fileSize -= stride;
        }

      } catch (Exception e) { e.printStackTrace(); }

    }
  }).start();

}
```

클라이언트로부터 #GET 명령어를 받을 때 호출된다. Client 클래스의 getFile 메서드와 대응되며, 작동 원리는 위 getFileFromClient 메서드와 유사하다. 우선 보낼 파일을 Server 클래스의 files 변수에서 찾아보고, 존재하지 않을 시 클라이언트에게 그렇다는 메시지와 더불어 -1을 보낸다. 위 설명한 getFileFromClient 메서드와 마찬가지로 getFile 메서드는 파일을 수신하기 전에 파일 크기에 대한 정보를 수신하기 때문이다. 그럼 getFile 메서드에서도 -1의 파일 크기를 수신하면 메서드를 종료하면 된다.

terminate 메서드

```
public void terminate() {
  if (members != null) leaveRoom();

  try {
    if(reader != null) reader.close();
    if(writer != null) writer.close();

    if(dataInputStream != null) dataInputStream.close();
    if(dataOutputStream != null) dataOutputStream.close();

  } catch (Exception e) { e.printStackTrace(); }

}
```

#EXIT 명령어는 기존 채팅방에서 퇴장할 때 사용될 뿐, 프로그램을 종료시키지는 않으므로 따로 #QUIT 명령어를 통해 클라이언트의 프로그램을 정상적으로 종료시킬 수 있도록 구현하였다. 이때 이 terminate 메서드를 호출시킴으로써, 우선 기존 채팅방에서 퇴장하고, 모든 IO Stream을 close 해준다.

run 메서드

```
public void run() {
  String input;
  try {
    while (true) {

      input = reader.readLine();

      if (input.charAt(0) == '#') {
        String[] userArgs = input.split(" ");

        if (userArgs[0].equals("#JOIN")) joinRoom(userArgs[1], userArgs[2]);
        else if (userArgs[0].equals("#CREATE")) createRoom(userArgs[1], userArgs[2]);

        else if (userArgs[0].equals("#PUT")) getFileFromClient(userArgs[1]);
        else if (userArgs[0].equals("#GET")) pushFileToClient(userArgs[1]);

        else if (userArgs[0].equals("#STATUS")) showStatus();
        else if (userArgs[0].equals("#EXIT")) leaveRoom();
        else if (userArgs[0].equals("#QUIT")) { terminate(); return; }

        else sendMessageToClient("### INVALID COMMAND: " + userArgs[0].substring(1));

      }
      else sendMessageToMembers("FROM " + username + ": " + input);
    }
  } catch (Exception e) { terminate(); return; }
}
```

ConnectionThread는 위 언급했듯이 Thread 클래스를 상속하므로, run 메서드를 재정의할 수 있다. 이 run 메서드는 클라이언트로부터의 입력을 기다리고, 그 입력에 적절한 메서드를 호출시킨

다. 프로그램을 종료할 때는 terminate 메서드 호출 뿐 아니라 return을 통해 run 메서드를 종료시킴으로써 이 쓰레드를 종료시킨다.

Client 클래스

멤버 변수

```
private Socket chatSocket;
private Socket fileSocket;
private Scanner scan;
private PrintWriter writer;
private BufferedReader reader;
private DataInputStream dataInputStream;
private DataOutputStream dataOutputStream;
```

Server 클래스와 마찬가지로 채팅 메시지 송수신에 대한 소켓 chatSocket, 파일 송수신에 대한 소켓 fileSocket을 갖는다. scan은 사용자의 키보드 입력을 읽는다. writer와 reader는 chatSocket을 통해 각각 메시지 송신, 수신을 맡는다. dataInputStream과 dataOutputStream은 fileSocket을 통해 각각 파일 수신, 송신을 맡는다.

생성자

```
public Client(InetAddress ip, int port1, int port2) {
  try {
    chatSocket = new Socket(ip, port1);
    fileSocket = new Socket(ip, port2);

    scan = new Scanner(System.in);
    reader = new BufferedReader(new InputStreamReader(chatSocket.getInputStream()));
    writer = new PrintWriter(new OutputStreamWriter(chatSocket.getOutputStream()));

    dataInputStream = new DataInputStream(fileSocket.getInputStream());
    dataOutputStream = new DataOutputStream(fileSocket.getOutputStream());

  } catch (Exception e) { e.printStackTrace(); }

}
```

인자로 받은 두 포트번호에 chatSocket과 fileSocket을 각각 바인딩해주고, 이들의 IO Stream을 통해 위 멤버변수들을 초기화해준다.

start 메서드

```
public void start() {
  // Send
  new Thread(new Runnable() {
    @Override
    public void run() {
      String message;
      while (true) {

        message = scan.nextLine();
        if (message.isBlank()) continue;

        writer.println(message);
        writer.flush();

        if (message.equals("#QUIT")) terminate();

        if (message.charAt(0) == '#') {
          String userArgs[] = message.split(" ");
          if (userArgs[0].equals("#PUT")) putFile(userArgs[1]);
          else if (userArgs[0].equals("#GET")) getFile(userArgs[1]);
        }

      }
    }
  }).start();

  // Receive
  new Thread(new Runnable() {
    @Override
    public void run() {
      String message;
      try {
        while (true) {
          message = reader.readLine();
          if (message == null) terminate();
          System.out.println(message);
        }
      } catch (Exception e) { terminate(); }
    }
  }).start();

}
```

Client의 메시지 송수신을 맡는 메서드이다. 위 쓰레드는 입력 및 송신, 아래는 수신을 맡는다. ConnectionThread가 단독으로 처리하는 #CREATE, #JOIN, #EXIT 명령어와 달리, #QUIT, #PUT, #GET 명령어는 Client 클래스에서도 처리함을 볼 수 있다.

putFile 메서드

```
public void putFile(String fileName) {

  new Thread(new Runnable() {
```

```
    @Override
    public void run() {
      FileInputStream fileInputStream;
      try {
        fileInputStream = new FileInputStream(fileName);
      } catch (FileNotFoundException e) {
        System.out.println("### FILE <" + fileName + "> DOES NOT EXIST");
        try { dataOutputStream.writeInt(-1); } // notify server to not wait for file
        catch (Exception ee) { ee.printStackTrace(); }
        return;
      }
      try {
        byte[] fileBytes = fileInputStream.readAllBytes();

        int fileSize = fileBytes.length;

        dataOutputStream.writeInt(fileSize);

        int offset = 0;
        int stride = 64000;

        System.out.println("\n### Uploading <" + fileName + "> (" + fileSize + " Bytes)...");
        System.out.print("### Progress : [");

        while (fileSize > 0) {
          int len = stride < fileSize ? stride : fileSize;
          dataOutputStream.write(fileBytes, offset, len);
          offset += stride;
          fileSize -= stride;
          System.out.print('#');
        }

        System.out.println("]\n### Successfully uploaded file <" + fileName +">\n");

        fileInputStream.close();

      } catch (Exception e) { e.printStackTrace(); }
    }
  }).start();
}
```

#PUT 명령어를 던졌을 때 호출되는 메서드이다. ConnectionThread 클래스의 pushFileToClient 메서드와 유사한데, 실제 파일을 읽어오기 위해 FileInputStream을 쓰고 송신하는 64000 바이트마다 진척도를 표시한다는 점에서 차이가 있다.

getFile 메서드

```
public void getFile(String fileName) {

  new Thread(new Runnable() {
    @Override
    public void run() {
      try {
        int fileSize = dataInputStream.readInt();
        if (fileSize == -1) return;
```

```
        byte[] fileBytes = new byte[fileSize];

        // This creates a file by the name fileName
        FileOutputStream fileOutputStream = new FileOutputStream(fileName);

        int offset = 0;
        int stride = 64000;

        System.out.println("\n### Downloading <" + fileName + "> (" + fileSize + " Bytes)...");
        System.out.print("### Progress : [");

        while (fileSize > 0) {
          int len = stride < fileSize ? stride : fileSize;
          dataInputStream.read(fileBytes, offset, len); // store at fileBytes
          fileOutputStream.write(fileBytes, offset, len); // copy contents in fileBytes to file
          offset += stride;
          fileSize -= stride;
          System.out.print('#');
        }

        fileOutputStream.close();

        System.out.println("]\n### Successfully downloaded file <" + fileName + ">\n");

      } catch (Exception e) { e.printStackTrace(); }
    }
  }).start();

}
```

#GET 명령어를 던졌을 때 호출되는 메서드이다. ConnectionThread 클래스의 getFileFromClient 메서드와 유사한데, ConnectionThread의 pushFileToClient 메서드에서 보내는 바이트 스트림을 실제 파일로 저장하기 위해 FileOutputStream을 쓰고 수신하는 64000 바이트마다 진척도를 표시한다는 점에서 차이가 있다.

terminate 메서드

```
public void terminate() {
  try {
    if(scan != null) scan.close();

    if(reader != null) reader.close();
    if(writer != null) writer.close();

    if(dataInputStream != null) dataInputStream.close();
    if(dataOutputStream != null) dataOutputStream.close();

    System.exit(0);
  } catch (Exception e) { e.printStackTrace(); }

}
```

ConnectionThread의 terminate 메서드와 마찬가지로, 클라이언트가 프로그램을 종료할 시 호출되는 메서드이다. 모든 IO Stream을 close 해주고, System.exit(0)을 통해 프로그램을 직접 종료시킨다. ConnectionThread는 Server 클래스의 inner 클래스이기 때문에, System.exit(0)을 호출하면 Server 프로그램이 종료되어버리지만 Client 클래스에선 문제가 없다.

main 메서드

```
public static void main(String[] args) throws Exception {

  InetAddress ip = InetAddress.getByName(args[0]);
  int port1 = Integer.parseInt(args[1]);
  int port2 = Integer.parseInt(args[2]);

  new Client(ip, port1, port2).start();

}
```

실행인자로 받아온 IP 주소와 두 포트번호를 사용해 Client 객체를 생성하고, 위 start 메서드를 호출한다.

---

2. 컴파일 및 실행 방법

소스 파일(Client.java, Server.java)이 위치한 경로에서 터미널을 실행시킵니다. 저는 윈도우즈 cmd창을 사용했습니다.

다음, 터미널에서 1. javac Server.java 2. javac Client.java 를 입력해 두 파일을 컴파일합니다.

다음, 프로그램을 실행시키기 위해 java Server (포트번호1) (포트번호2)를 입력합니다. Client를 실행하기 위해서는 Server 프로그램이 먼저 실행 중이어야 합니다.

다음, 다른 터미널(마찬가지로 소스 파일이 위치한 경로에서)을 실행시킨 뒤 java Client (IP) (포트번호1) (포트번호2)를 입력합니다.

프로그램을 종료하려면 #QUIT을 입력하면 됩니다.

#PUT (파일명) 명령어를 통해 보낼 이 파일 역시 소스 파일이 위치한 경로에 존재해야 합니다.

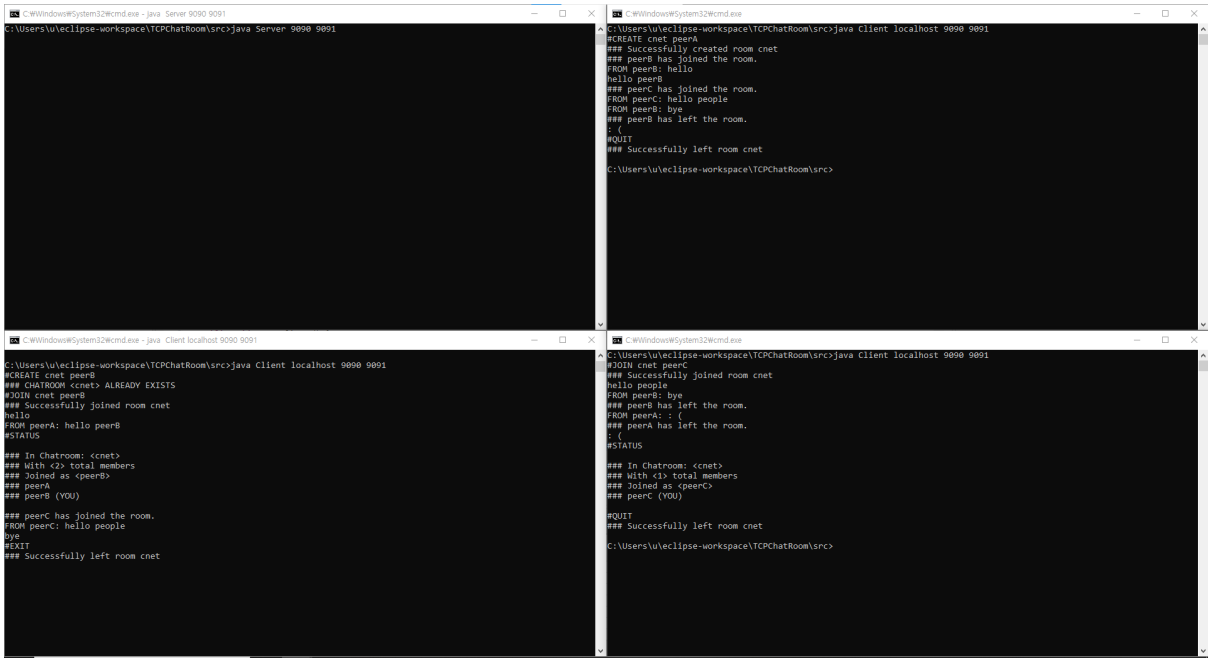
마찬가지로 #GET (파일명) 명령어를 통해 받아온 파일도 소스 파일이 위치한 경로에 생성됩니다.

#PUT 과 #GET이 잘 작동하는지 확인하기 위해, #PUT 명령어를 입력한 뒤 실제 파일을 제거해봅니다.

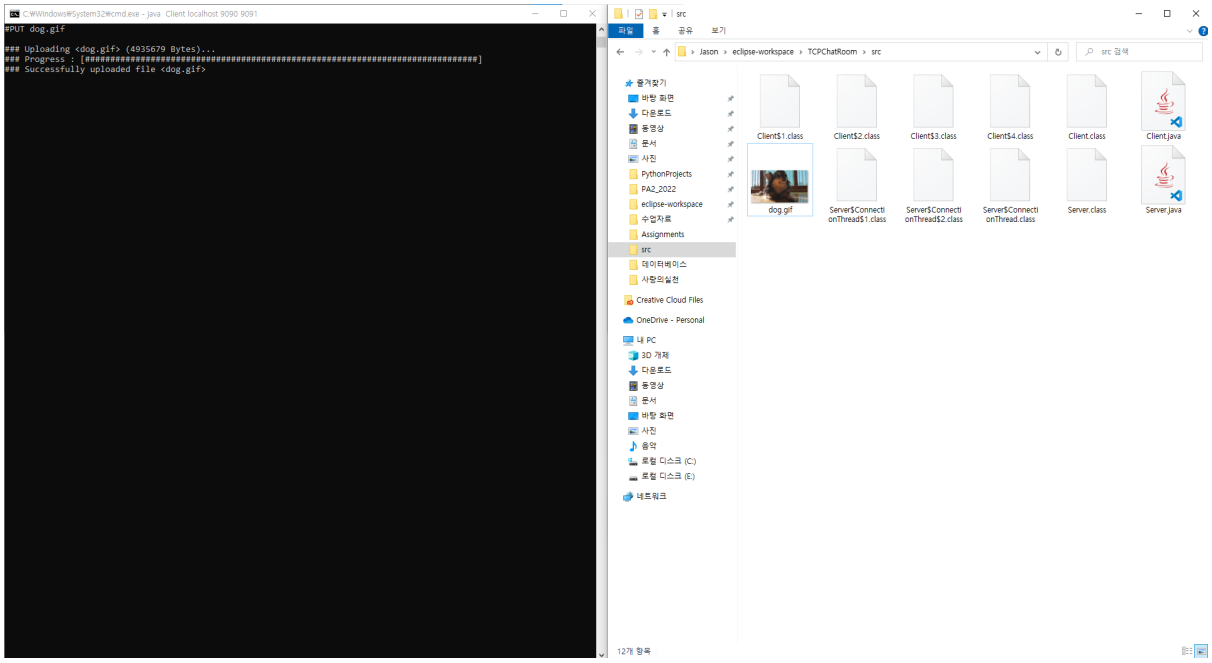
다음 #GET 명령어를 입력하면 다시 그 파일이 생성됩니다.

---

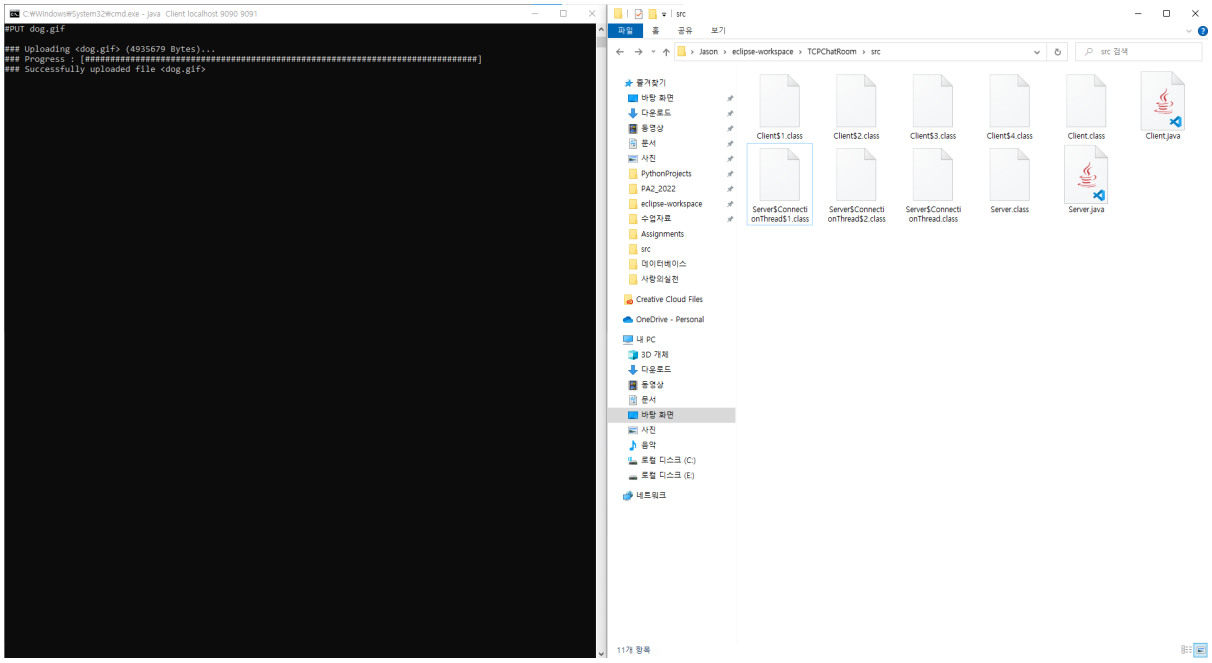
3. 예시 실행 화면



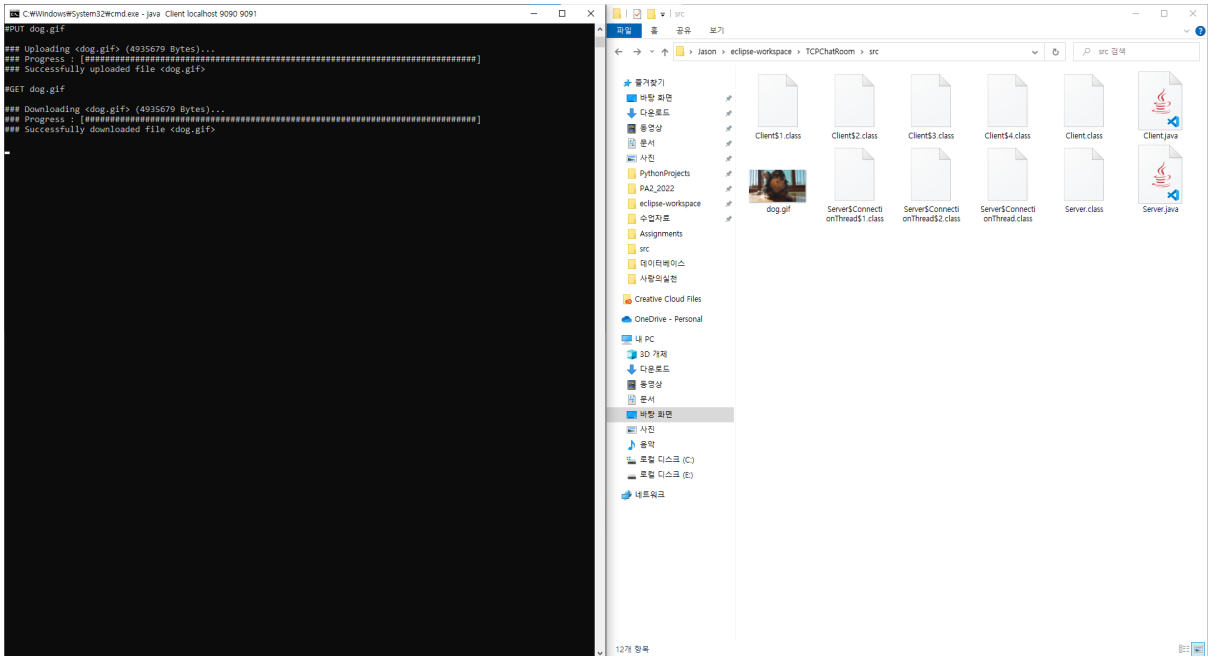
채팅



#PUT dog.gif 입력



dog.gif 삭제



#GET dog.gif을 입력하면 dog.gif가 다시 생성된다.