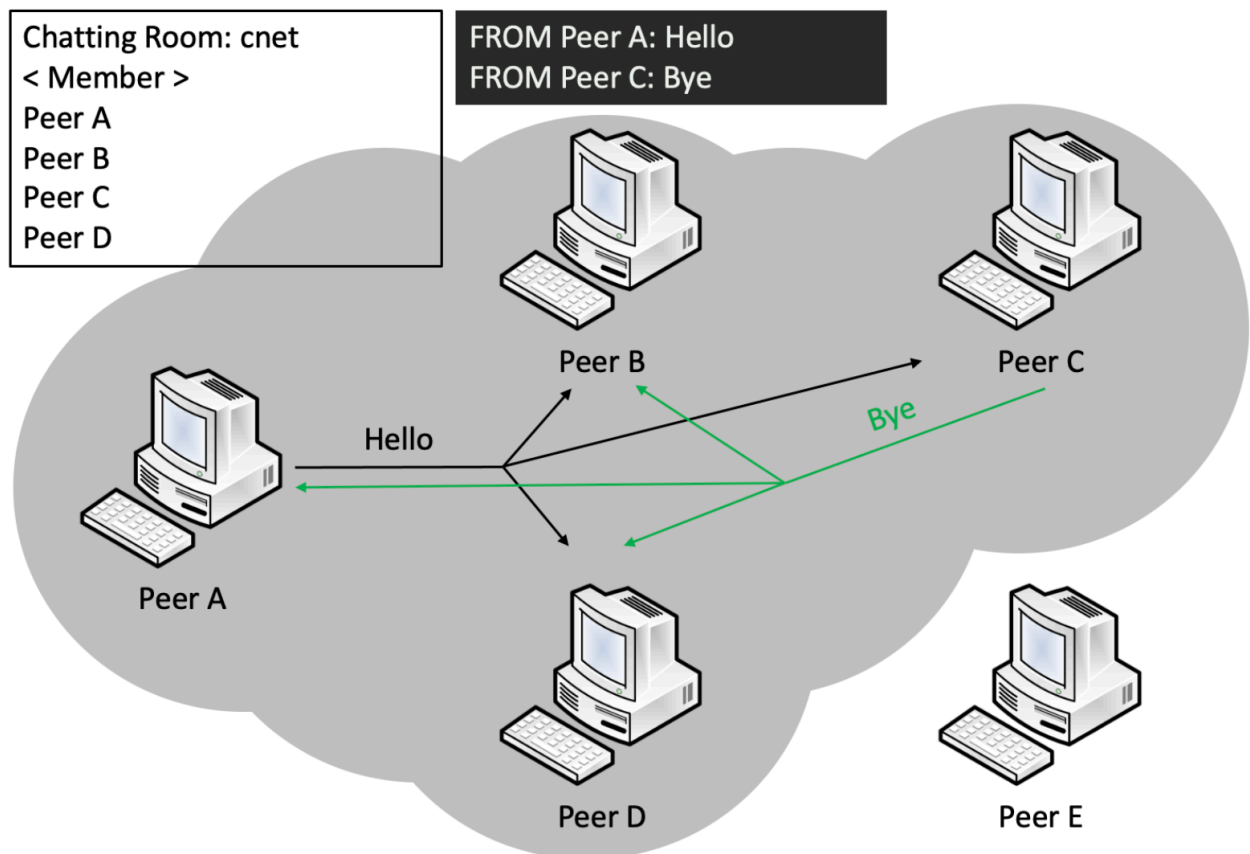

UDP Multicast Chatting

디자인 및 구현 문서

2019040591 박진수



소스 파일 구성 - ./src

- Peer.java

최초 실행되는 코드로, 명령어를 입력받고, UDPMulticastReceiver와 UDPMulticastSender 클래스를 멀티스레드로 호출한다.

- UDPMulticastReceiver.java

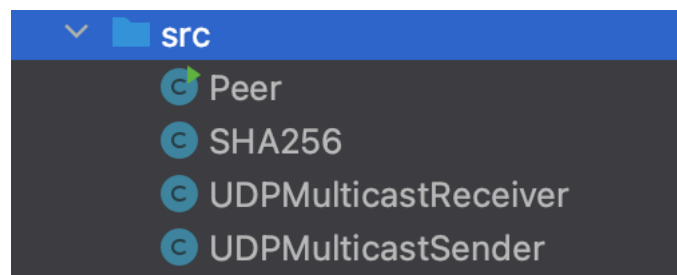
메시지를 수신하는 기능을 담당하는 코드이다.

- UDPMulticastSender.java

메시지를 입력받고, 전송하는 기능을 담당하는 코드이다.

- SHA256.java

입력받은 채팅방 이름을 “SHA-256” 해시를 이용해서 Multicast address 225.x.y.z 로 변환하여 반환하는 코드이다.



컴파일 방법

1. 터미널 창에서 Assignment1_2019040591_박진수 폴더로 이동한다.

실행 예시: `cd Assignment1_2019040591_박진수`

```
jinsoo@parkjinsoo-MacBookAir 컴퓨터 네트워크 % cd Assignment1_2019040591_박진수
jinsoo@parkjinsoo-MacBookAir Assignment1_2019040591_박진수 %
```

2. `javac src/*.java -d bin -encoding utf-8` 을 입력하여 bin폴더 생성 후 그 안에 class 파일들을 생성한다.

실행 예시: `javac src/*.java -d bin -encoding utf-8`

```
jinsoo@parkjinsoo-MacBookAir Assignment1_2019040591_박진수 % javac src/*.java -d bin -encoding utf-8
Note: src/UDPMulticastReceiver.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

3. class 파일들이 있는 bin 폴더로 이동한다.

실행 예시: `cd bin`

```
jinsoo@parkjinsoo-MacBookAir Assignment1_2019040591_박진수 % cd bin
jinsoo@parkjinsoo-MacBookAir bin %
```

실행 방법

1. 채팅 프로그램 실행

프로그램 실행 인자로 (port number)를 입력받는다.

- 실행 예시: java Peer 8081

```
jinsoo@parkjinsoo-MacBookAir bin % java Peer 8081
채팅방 입장 방법 : #JOIN (참여할 채팅방의 이름) (사용자 이름)
```

2. 채팅 연결

채팅방에 참여하기 위해 다음과 같은 명령어를 입력받는다.

#JOIN (참여할 채팅방의 이름) (사용자 이름)

- 실행 예시: #JOIN room jinsoo

```
#JOIN room jinsoo
room 채팅방에 입장 하였습니다 !
```

3. 메시지 입력

채팅방 입장 후 메시지를 입력받는다.

- 실행 예시: this is test message

```
room 채팅방에 입장 하였습니다 !
this is test message
jinsoo: this is test message
```

4. 채팅방 나가기

채팅방에서 나가기 위해 다음과 같은 명령어를 입력받는다.

- 실행 예시: #EXIT

```
#EXIT
```

```
이 채팅방을 나갑니다...
```

```
-- jinsoo 님이 채팅을 종료하셨습니다. --
```



코드 디자인 및 설명 - Peer.java

8번째 줄

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

프로그램 실행시 최초 실행되는 main 함수로, 프로그램 실행시 함께 입력된 포트번호를 매개변수로 받는다.

16~21번째 줄

```
if (args.length != 1) {  
    System.out.println("Port 번호를 올바르게 입력해주세요.");  
    System.exit(0);  
} else {  
    portNum = Integer.parseInt(args[0]);  
}
```

args의 길이가 1이 아니면, 즉 포트번호가 실행 시 제대로 입력되지 않았으면 프로그램을 종료하는 부분이다. 제대로 입력받았으면, portNum 변수에 저장한다.

26~32번째 줄

```
while ((inputString = reader.readLine()) != null) {  
    reader = new BufferedReader(new  
InputStreamReader(System.in));  
    // 아무 입력 안받으면 다시 입력 받기  
    if (inputString.length() == 0) {  
        continue;  
    }  
    String[] splitedInput = inputString.split(" ");
```

inputString 변수에 입력받은 내용을 집어넣고, 빈 입력이 아닌 정상적인 입력이라면, 공백(" ")을 기준으로 해당 문자열을 나누어(split) splitedInput 배열에 담는다. 위 코드 위치는 #JOIN (채팅방 이름) (사용자 이름) 입력이 예상되는 부분이다. 따라서 splitedInput[0]에는 "#JOIN"이 담기게 된다.

35번째 줄

```
isCommand = (splitedInput[0].charAt(0) == '#') ? true : false;
```

splitedInput[0]의 첫 번째 char를 확인해 '#'으로 시작하는지 확인한다. 즉 명령어의 형태를 띄고 있는지 확인한 후, 맞다면 isCommmand 변수를 true로 변경한다.

38~49번째 줄

```
// commandPhrase는 #지운 명령어 자체(e.g. #JOIN -> JOIN)
commandPhrase = splittedInput[0].substring(1);

switch (commandPhrase) {
    case "JOIN":

        // 인풋 다음과 같이 받음. #JOIN (참여할 채팅방의 이름) (사용자 이름)
        if (splittedInput.length != 3) {
            System.out.println("ERROR! 다음과 같은 형식으로 입력해주세요.");
            System.out.println("#JOIN (참여할 채팅방의 이름) (사용자 이름)");
            continue;
        }
}
```

splittedInput[0]의 값을 확인하여, 첫 번째 char가 '#'인 경우, '#'을 제외한 문자열을 commandPhrase 변수에 저장한다. ("#JOIN"의 경우 "JOIN"이 담긴다. 코드의 확장성을 고려하여 switch문을 이용해, #JOIN이 아닌 다른 명령어가 생길 경우도 고려하여 코드를 작성하였다. 아래의 if문을 이용하여, input이 "#JOIN (채팅방 이름) (사용자 이름)"의 형식으로 입력된 경우만 아래의 코드로 계속 진행하도록 하였다.

50~51번째 줄

```
String roomName = splittedInput[1];
String userName = splittedInput[2];
```

splittedInput[1]에는 (채팅방 이름), splittedInput[2]에는 (사용자 이름)이 담기게 되는데 각각을 roomName, userName 변수에 저장한다.

52~54번째 줄

```
SHA256 sha256 = new SHA256();
// String형의 multicastAddress를 InetAddress형으로 바꿈.
InetAddress multicastAddress =
    InetAddress.getByName(sha256.getMulticastAddress(roomName));
```

미리 만들어놓은 SHA256클래스의 인스턴스를 생성한 후, 해당 인스턴스의 getMulticastAddress 메소드를 호출한다. getMulticastAddress 메소드는 roomName 변수를 인자로 받아 SHA-256 해시를 이용해 "225.x.y.z" 형태의 MulticastAddress 문자열을 반환한다. 그리고 InetAddress의 getByName 메소드를 이용하여, 문자열 끝의 IP 주소를 이용해 multicastAddress이라는 InetAddress 인스턴스를 초기화 한다.

56~66번째 줄

```
try {
    udpMulticastReceiver = new
UDPMulticastReceiver(multicastAddress, portNum, userName);
    udpMulticastReceiver.start();
    udpMulticastSender = new UDPMulticastSender(multicastAddress,
portNum, userName, roomName);
    udpMulticastSender.start();
    udpMulticastSender.join();
    udpMulticastReceiver.stopThread();
    udpMulticastReceiver.join();
} catch (Exception e) {
    e.printStackTrace();
}
```

메시지 수신을 담당하는 `udpMulticastReceiver` 인스턴스와, 메시지 발신을 담당하는 `udpMulticastSender` 인스턴스를 생성한다. 위 두 클래스 인스턴스는 `Thread`를 상속받았으므로, `start()` 메소드를 통해 두 클래스를 시작시킨다. `join()` 메소드를 이용하여 위 두 스레드가 모두 종료된 후, 메인함수의 코드가 재개 되도록 하였다. 입력으로 "#EXIT"를 받으면, `udpMulticastSender` 스레드가 종료되고, 이후 `stopThread()` 메소드가 실행되어 `udpMulticastReceiver`의 `exitThread` flag가 `true`가 되도록 하여 `udpMulticastReceiver` 스레드 또한 종료하도록 유도하였다. 이후 다시 메인함수의 `while`문 처음으로 돌아가 명령어 입력을 기다린다.

코드 디자인 및 설명 - UDPMulticastSender.java

6번째 줄

```
public class UDPMulticastSender extends Thread {
```

UDPMulticastSender class는 Thread class 를 상속받아, main함수 상에서 UDPMulticastSender와 UDPMulticastReceiver 스레드가 동시에 동작할 수 있도록 했다.

16~21번째 줄

```
public UDPMulticastSender(InetAddress inputInetAddress, int
inputPortNum, String inputUserName, String inputRoomName) {
    inetAddress = inputInetAddress;
    portNumber = inputPortNum;
    userName = inputUserName;
    roomName = inputRoomName;
}
```

위는 UDPMulticastSender 클래스의 생성자로, IP주소를 담은 InetAddress 변수, 포트번호, 채팅방에 참여한 유저의 이름, 그리고 참여하는 채팅방의 이름을 인자로 받아 각각에 대응하는 클래스의 인스턴스 변수를 초기화 한다.

23번째 줄

```
public void run() {
```

메인함수에서 UDPMulticastSender.start()를 실행하면, 위의 run() 메소드가 실행된다.

26번째 줄

```
datagramSocket = new DatagramSocket();
```

데이터(datagram)의 발신을 담당하는 소켓 인스턴스를 생성하여 datagramSocket을 초기화한다.

28~29번째 줄

```
while (true) {
    byte[] chunk = new byte[512];
```

채팅방에 입장한 후 메시지를 계속 입력받아야 하므로, while(true)를 이용해 계속 입력을 반복하여 받도록 하였고, 512바이트 크기의 chunk라는 byte array는 채팅 메시지를 chunk 단위(512 byte)로 나누어서 전송하는데 필요한 변수이다.

while문 안에 해당 코드를 작성한 이유는, chunk에 그 다음 데이터들이 덮어쓰워진 채로 전송될 위험을 방지하기 위함이다.

31~44번째 줄

```
String tempInput = bufferedReader.readLine();
if (tempInput.charAt(0) == '#') {
    if (tempInput.equals("#EXIT")) {
        System.out.println("이 채팅방을 나갑니다...");
        messageToSend = tempInput;
        String packedMessage = userName + ": " + messageToSend;
        System.arraycopy(packedMessage.getBytes(), 0, chunk, 0,
packedMessage.getBytes().length);
        datagramPacket = new DatagramPacket(chunk, chunk.length,
inetAddress, portNumber);
        datagramSocket.send(datagramPacket);
        break; // while문 나가기
    } else {
        System.out.println("잘못된 명령어가 입력되었습니다.");
        System.out.println("다시 입력해주세요.");
    }
}
```

입력받은 내용(보낼 메시지)을 tempInput 변수에 저장하도록 하였고, “#EXIT”가 입력되었다면 채팅방을 나가도록 구현하기 위해 if문을 이용했다. 채팅방을 나갈 때, 다른 이용자들에게도 채팅방을 나갔다는 소식을 전파하기 위해, datagramSocket을 통해, “(이용자 이름): #EXIT”의 문자열 데이터를 전달한다. 해당 메시지를 유저는 메시지 입력시, ‘#’가 들어간 문자열은 오로지 명령어로만 사용할 수 있기 때문에, ‘#’이 들어간 메시지는 따로 명령어로 취급하여 처리한다. 그리고 “#EXIT”후 해당 내용을 다른 유저들에게 전파한 이후, break문을 통해 while문을 빠져나가 finally에 해당하는 코드를 실행하여 열려있는 소켓을 닫고, 해당 스레드를 종료하게 된다.

45~41번째 줄

```
} else { // 명령어 입력이 아닌 메시지 입력의 경우 처리
    messageToSend = tempInput;
    String packedMessage = userName + ": " + messageToSend;
    if (packedMessage.getBytes().length <= 512) { // byte가 512바이트 이하이면
        System.arraycopy(packedMessage.getBytes(), 0, chunk, 0,
packedMessage.getBytes().length);
        datagramPacket = new DatagramPacket(chunk, chunk.length,
inetAddress, portNumber);
        datagramSocket.send(datagramPacket);
    }
```

만약 입력이 ‘#’로 시작하지 않는 문자열이라면(일반 메시지 전송), 해당 메시지를 “(유저 이름): (입력받은 문자열)”형태의 문자열로 만들어 packedMessage 변수

에 저장한다. 이후 해당 문자열을 바이트형태로 변환했을 때 해당 바이트배열의 길이를 구해, 512바이트 이하인 경우와 512바이트를 초과하는 경우로 나누어서 처리한다. 512바이트 이하인 경우에는 512바이트 길이의 chunk에 한번에 담아 전송할 수 있으므로, chunk 에 packedMessage의 데이터를 복사한 후 datagramPacket에 chunk를 담아 datagramSocket을 통해 전송한다.

52~64번째 줄

```
} else { // 512바이트 보다 큰 배열이면
    byte[] bigByteArr = new byte[packedMessage.getBytes().length];
    // 512바이트보다 큰 바이트 배열을 선언한다.
    System.arraycopy(packedMessage.getBytes(), 0, bigByteArr, 0,
        packedMessage.getBytes().length);
    int offset = 0;
    datagramPacket = new DatagramPacket(bigByteArr, offset, 512,
        inetAddress, portNumber);
    int bytesSent = 0;
    while (bytesSent < bigByteArr.length) {
        datagramSocket.send(datagramPacket);
        bytesSent += datagramPacket.getLength();
        int bytesToSend = bigByteArr.length - bytesSent;
        int size = (bytesToSend > 512) ? 512 : bytesToSend;
        datagramPacket.setData(bigByteArr, bytesSent, size);
    }
}
```

이전 부분에서 생성된 packedMessage를 바이트로 변환한 크기가 512바이트보다 클 경우, 데이터를 여러번에 걸쳐 나누어 전송해야 한다. chunk의 크기가 512바이트이므로, 일단 packedMessage를 바이트로 변환한 byte array를 bigByteArr 변수에 저장한 뒤, 처음에는 0~511번째 인덱스 (512개)에 해당하는 바이트들을 datagramPacket에 담아 전송한 후, while문을 통해 그 다음 잔여 데이터를 모두 보낼 때 까지 위와 유사한 과정을 반복한다.(512~1023, ...) 위의 과정을 반복하다 보내야 될 마지막 chunk를 보내야 하는 시점에서는 잔여 byte array 크기 만큼의 데이터를 datagramPacket에 담아 전송한다.

74~77번째 줄

```
} finally {
    if (!datagramSocket.isClosed())
        datagramSocket.close();
}
```

입력으로 "#EXIT"가 들어오면, break문을 거쳐 해당 위치로 이동하게 된다. 이후 datagramSocket을 닫고 해당 스레드를 종료한다.

코드 디자인 및 설명 - UDPMulticastReceiver.java

6번째 줄

```
public class UDPMulticastReceiver extends Thread {
```

UDPMulticastReceiver class는 Thread class 를 상속받아, main함수 상에서 UDPMulticastSender와 UDPMulticastReceiver 스레드가 동시에 동작할 수 있도록 했다.

16~21번째 줄

```
public UDPMulticastReceiver(InetAddress inputInetAddress, int
inputPortNum, String inputUserName) {
    inetAddress = inputInetAddress;
    portNumber = inputPortNum;
    userName = inputUserName;
    exitThread = false;
}
```

위는 UDPMulticastReceiver 클래스의 생성자로, IP주소를 담은 InetAddress 변수, 포트번호, 그리고 채팅방에 참여한 유저의 이름을 인자로 받아 각각에 대응하는 클래스의 인스턴스 변수를 초기화 한다. 그리고 위 스레드를 종료하기 위한 flag로 사용하는 exitThread 변수를 false로 초기화 시킨다.

23~25번째 줄

```
public void stopThread() {
    exitThread = true;
}
```

main함수에서 해당 클래스의 stopThread() 메서드를 수행하면, flag기능을 하는 인스턴스 변수 exitThread의 값을 true로 변경한다.

27번째 줄

```
public void run() {
```

메인함수에서 UDPMulticastReceiver.start()를 실행하면, 위의 run() 메소드가 실행된다.

28~30번째 줄

```
try {
    multicastSocket = new MulticastSocket(portNumber);
    multicastSocket.joinGroup(inetAddress);
```

메시지 수신을 담당하는 `multicastSocket`을 만들고, `multicast` 주소인 `inetAddress`에 해당하는 `multicast group`에 해당 소켓을 참여시킨다.

31~37번째 줄

```
while (true) {
    if (exitThread) break;
    // 채팅 메시지를 chunk 단위(512 byte)로 나누어서 수신
    byte[] chunk = new byte[512];
    datagramPacket = new DatagramPacket(chunk, chunk.length);
    multicastSocket.receive(datagramPacket);
    String receivedMessage = new
String(datagramPacket.getData()).trim();
```

해당 스레드는 계속 메시지를 수신해야 하므로 `while`문을 이용하였고, `exitThread` flag의 값이 `true`가 되면 `break`하여 해당 반복문을 빠져나와 스레드를 종료하도록 하였다. 수신 데이터를 담는 역할인 512바이트 크기의 `chunk` 변수를 만들고, `datagramSocket`을 통해 들어온 담긴 값을 `datagramPacket`을 통해 `chunk`가 전달받도록 했다. `byte array`에 들어온 '`\u0020`' 이하의 불필요한 값들을 제거하기 위해 `trim()` 메서드를 사용했다.

38~52번째 줄

```
String[] colonDivided = receivedMessage.split(":");
    if (colonDivided.length == 2) {
        String senderName;
        String sendContent;
        senderName = colonDivided[0].strip();
        sendContent = colonDivided[1].strip();
        if (sendContent.equals("#EXIT")) {
            System.out.println("-- " + senderName + " 님이 채팅을 종료하셨습니다. --");
        } else {
            System.out.println(receivedMessage);
        }
    } else { // 512바이트 초과인 데이터의 첫 번째 chunk를 제외한 나머지 부분이 들어오는 경우
        System.out.println(receivedMessage);
    }
}
```

수신한 메시지에 해당하는 문자열을 `:` 을 기준으로 나누었다. 이유는 사용자가 `"#EXIT"`를 입력했을 때 다른 유저들에게 해당 유저의 퇴장 소식을 전달하기 위함이었다. `':'` (`colon`)을 기준으로 앞에 오는 값이 (유저이름), 뒤에 오는 값이 (전송된 메시지) 일 것이라고 생각하고 코드를 작성했다. 다만 예외적으로 512바이트 초과된 문자열이 들어오는 경우 중에서, 첫번째 `chunk` 이후의 `chunk`에서 데이터에 `':'`가 입력된 경

우, ‘:’을 기준으로 앞과 뒤를 각각 (유저이름)과 (전송된 메시지)로 오인하는 경우가 발생하지만, 현재까지 주어진 조건에서는 “#EXIT”가 입력된 경우만 그 둘을 구분하므로 문제의 가능성은 없었다. 다만 추후 코드에 추가적인 기능을 추가해야 하는 경우, 이 문제를 고려해야 할 것이다.

56~59번째 줄

```
} finally {  
    if (!multicastSocket.isClosed()) {  
        multicastSocket.close();  
    }  
}
```

유저가 “#EXIT”를 입력한 경우, exitThread flag의 값이 true로 바뀌고 이에 따라 break문이 실행되어 while문을 빠져나간다. 이후 socket을 닫고 스레드를 종료한다.

코드 디자인 및 설명 - SHA256.java

7~11번째 줄

```
public String getMulticastAddress(String text) throws
NoSuchAlgorithmException {
    MessageDigest md = MessageDigest.getInstance("SHA-256");
    md.update(text.getBytes());
    return bytesToMulticastAddress(md.digest());
}
```

“SHA-256” 해시 알고리즘을 이용하는 MessageDigest 인스턴스를 만든 후, 매개 변수로 받은 채팅방 이름을 byte array로 변환한 후 이를 이용하여 해당 digest 인스턴스를 업데이트 한다. md.digest() 메서드를 수행하면 입력받은 문자열에 해당하는 바이트 array가 “SHA-256” 알고리즘을 통해 해싱되는데, 이를 다시 bytesToMulticastAddress 메서드의 인자로 집어넣는다.

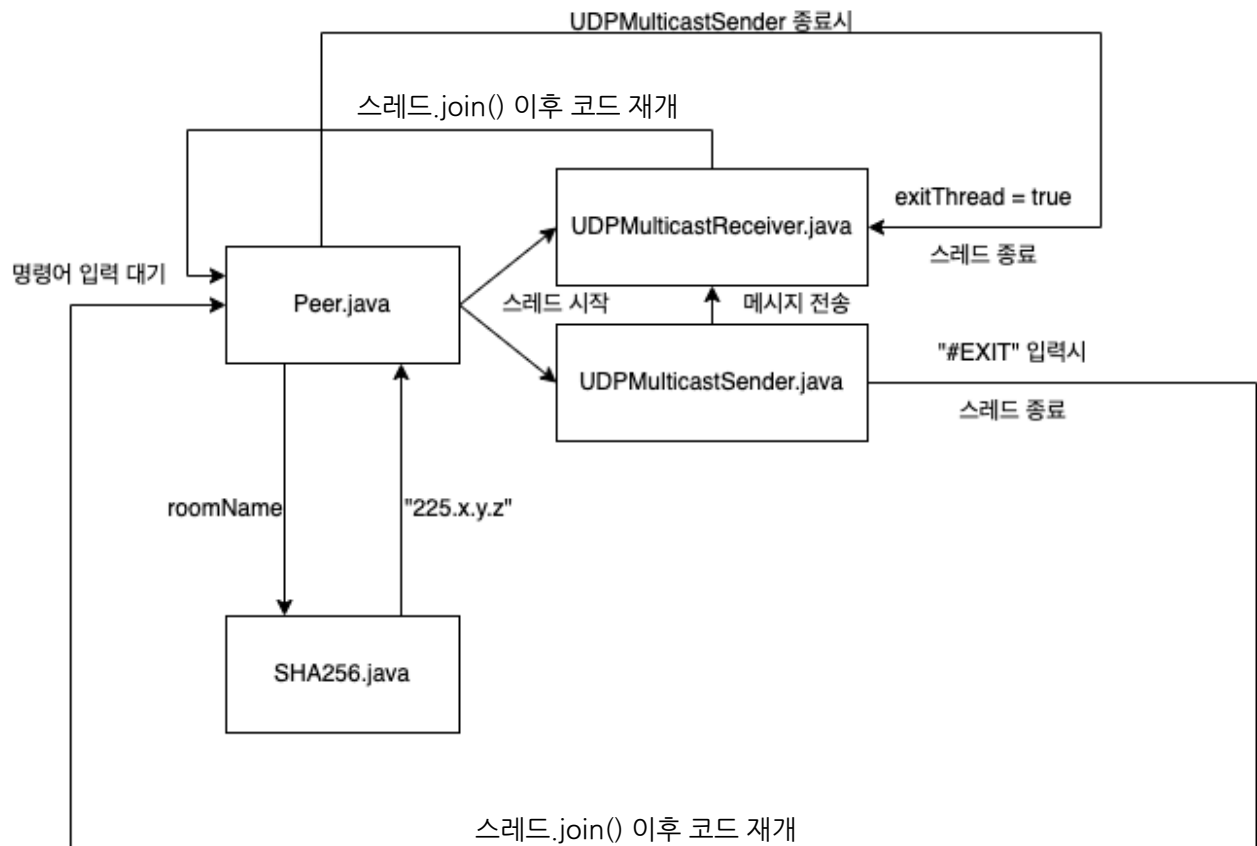
13~24번째 줄

```
private String bytesToMulticastAddress(byte[] bytes) {
    //225.x.y.z 로 변환 후 리턴
    StringBuilder stringBuilder = new StringBuilder();
    int lengthOfBytes = bytes.length;
    stringBuilder.append("225.");
    for (int i = 0; i < 3; i++) {

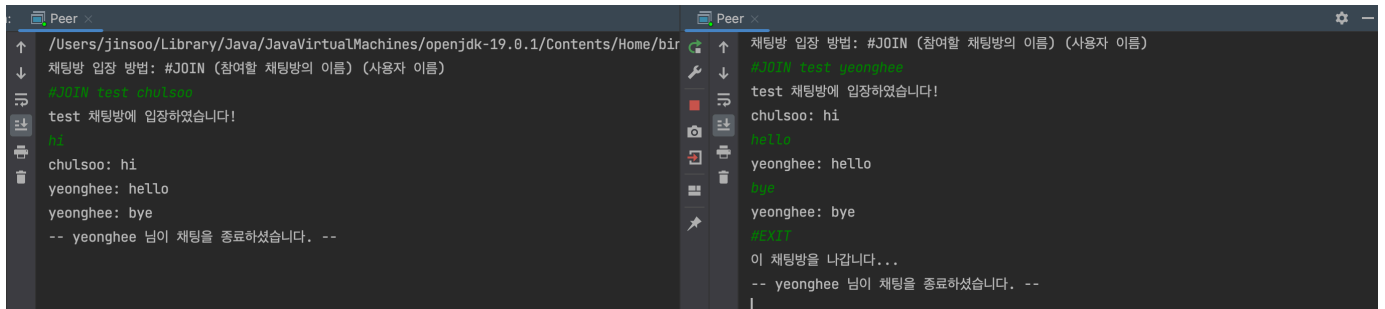
        stringBuilder.append(Byte.toUnsignedInt(bytes[lengthOfBytes-3+i]))
        ;
        stringBuilder.append(".");
    }
    String multicastAddress =
    stringBuilder.substring(0, stringBuilder.length()-1);
    return multicastAddress;
}
```

인자로 전달받은 해싱된 byte array들을 매개변수로 받아 마지막 3개의 바이트를 unsignedInt 형으로 변환하여 “225.x.y.z” 꼴의 문자열로 만든 후 해당 문자열을 반환한다.

코드 디자인 요약



예시 실행 화면



```
Peer <
/Users/jinsoo/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin
채팅방 입장 방법: #JOIN (참여할 채팅방의 이름) (사용자 이름)
#JOIN test chulsoo
test 채팅방에 입장하였습니다!
hi
chulsoo: hi
yeonghee: hello
yeonghee: bye
-- yeonghee 님이 채팅을 종료하였습니다. --

Peer <
채팅방 입장 방법: #JOIN (참여할 채팅방의 이름) (사용자 이름)
#JOIN test yeonghee
test 채팅방에 입장하였습니다!
chulsoo: hi
hello
yeonghee: hello
bye
yeonghee: bye
#EXIT
이 채팅방을 나갑니다...
-- yeonghee 님이 채팅을 종료하였습니다. --
```