# Multi-threading 기반 채팅 프로그램 기능 향상

컴퓨터 네트워크 (1258)
22311947 김서현

## 1. 소개

본 프로젝트는 수업 시간에 제공된 “chat_serv.c”와 “chat_clnt.c” 코드를 기반으로 하여, Multi-threading 기반 채팅 프로그램의 기능을 향상한다.

기능 향상 이전의 기본 “chat_serv.c”와 “chat_clnt.c” 코드는 멀티 쓰레드 방식의 채팅 서버를 운영한다. 서버는 한 번에 여러 클라이언트와 통신하며 클라이언트는 오직 한 번에 한 개의 서버와만 통신할 수 있다.

“chat_serv.c” 코드에서 서버는 새 클라이언트가 접속할 때마다 새로운 쓰레드를 생성한다. 생성된 쓰레드에서는 특정 클라이언트의 메시지를 받아 서버에 저장된 전체 클라이언트에게 전송한다. 이 과정에서 사용되는 전역 변수는 임계 영역을 통해 안전하게 관리된다.

“chat_clnt.c” 코드에서 클라이언트는 서버와 접속 후 두 개의 쓰레드를 생성한다. 하나의 쓰레드는 읽기 전용, 다른 하나는 쓰기 전용으로 동작된다. 클라이언트는 하나의 프로그램 안에서 별도의 쓰레드를 사용함으로써 blocking 없이 읽기와 쓰기를 동시에 수행할 수 있다.

이번 프로젝트에서는 “chat_serv.c”와 “chat_clnt.c” 코드를 수정해 1:1 대화 기능을 구현한다. “chat_clnt.c”의 경우, 메시지를 전송하는 send_msg() 함수를 수정해 1:1 대화를 처리할 수 있도록 코드를 변경하였다. 클라이언트가 전송하려는 메시지의 가장 앞에 ‘@’가 붙어 있다면 ‘@receiver [name] text’의 형식으로 서버에게 데이터를 보내도록 하였다.

“chat_serv.c”의 경우, 1:1 대화를 위한 send_whisper_msg()라는 별도의 함수가 작성되었다. 서버가 클라이언트에게서 ‘@receiver [name] text’라는 포멧을 가진 데이터를 수신받으면 send_whisper_msg()가 실행된다. send_whisper_msg()는 클라이언트가 보낸 데이터에서 수신자의 이름(receiver)를 확인하고 해당 이름을 가진 클라이언트가 존재한다면 ‘(whisper) [name] text’라는 형식으로 해당 클라이언트에게 메시지를 전달한다.

또한 서버가 각 클라이언트의 이름을 알고 이를 char clnt_names[MAX_CLNT][NAME_SIZE];로 선언된 ‘clnt_names’라는 전역 변수에 저장하도록 하기 위해 사전에 데이터를 주고받는 과정을 추가하였다. 클라이언트가 서버에 연결된 직후, write()를 통해 연결된 서버에게 자신의 이름을 전달한다. 서버는 read()를 통해 클라이언트

의 이름을 전달받는다. 이후 클라이언트 소켓을 소켓 배열에 저장할 때 클라이언트의 이름도 함께 이름 배열에 저장한다. 두 배열 모두 뮤텍스를 통해 임계 영역 내에서 안전하게 관리된다.

## 2. 코드

해당 문서에서는 “chat_serv.c”와 “chat_clnt.c”의 전체 프로그램 코드에서 기능 향상을 위해 수정된 내용을 위주로 소개된다.

### 1) chat_serv.c

chat_serv.c 코드에서는 브로트캐스트 통신을 위한 send_msg() 메소드의 아래 1:1 통신을 위한 send_whisper_msg() 메소드가 추가되었다. 아래는 쓰레드가 실행하는 메소드인 handle_clnt() 메소드의 내용 중 일부이다.

```
while ((str_len = read(clnt_sock, msg, sizeof(msg))) != 0) {
        if (msg[0] == '@') {
                check = send_whisper_msg(msg, str_len);
                if (check == -1) write(clnt_sock, error, strlen(error));
        }
        else
                send_msg(msg, str_len);

        memset(msg, 0, sizeof(msg));
}
```

해당 코드는 클라이언트의 메시지를 수신한다. msg 배열 안에 담긴 클라이언트의 메시지가 ‘@’로 시작할 경우, send_whisper_msg() 메소드가 해당 메시지를 처리한다. send_whisper_msg()는 성공적으로 처리가 완료되었을 경우 0을 리턴한다.

만약 해당 1:1 메시지를 수신받을 클라이언트가 존재하지 않을 경우, send_whisper_msg()는 -1을 리턴한다. 이 경우 클라이언트는 메시지를 송신한 클라이언트에게 에러 메시지를 반환한다.

아래 코드는 1:1 메시지를 처리하는 send_whisper_msg()의 일부이다.

```
int i;
char* receiver, * text;
char whisper_msg[NAME_SIZE+BUF_SIZE]; //수신자에게 전달된 최종
                                        메시지를 담는 배열

receiver = strtok(msg, " ");
receiver++;
```

```
text = strtok(NULL, "");
```

서버는 자신이 전송받은 @receiver [name] text' 포멧의 메시지를 'receiver'와 '[name] text'으로 나눈다. 이후 임계 영역 내에서 해당 메시지를 수신받을 리시버를 찾는다.

```
pthread_mutex_lock(&mutx); //임계 영역 시작
if (!strcmp(receiver, "all")) {
        sprintf(whisper_msg, "%s %s", "(all)", text);
        for (i = 0; i < clnt_cnt; i++)
                write(clnt_socks[i], whisper_msg, strlen(whisper_msg));
        pthread_mutex_unlock(&mutx); //임계 영역 끝
        return 0; //메소드 정상 종료
}

for (i = 0; i < clnt_cnt; i++)
        if (!strcmp(clnt_names[i], receiver)) {
                sprintf(whisper_msg, "%s %s", "(whisper)", text);
                write(clnt_socks[i], whisper_msg, strlen(whisper_msg));
                pthread_mutex_unlock(&mutx); //임계 영역 끝
                return 0; //메소드 정상 종료
        }
```

리시버의 이름이 "all"일 경우, 모든 클라이언트에게 (all) 키워드를 붙인 메시지를 전송한다. 리시버의 이름이 clnt_names 배열에 존재할 경우, 해당 클라이언트에게 (whisper) 키워드를 붙인 메시지를 전달한다. 리시버의 이름이 clnt_names 배열에 존재하지 않을 경우, 임계 영역을 종료하고 -1을 리턴한다.

### 2) chat_clnt.c

chat_clnt.c 코드에서는 메시지를 송신하는 send_msg 함수가 일부 수정되었다.

```
if(msg[0] == '@')
{
        char* receiver, * text;
        memset(name_msg, 0, sizeof(name_msg));
        receiver = strtok(msg, " ");
        text = strtok(NULL, "");
        sprintf(receiver_name, "%s %s", receiver, name);
        sprintf(name_msg, "%s %s", receiver_name, text);
```

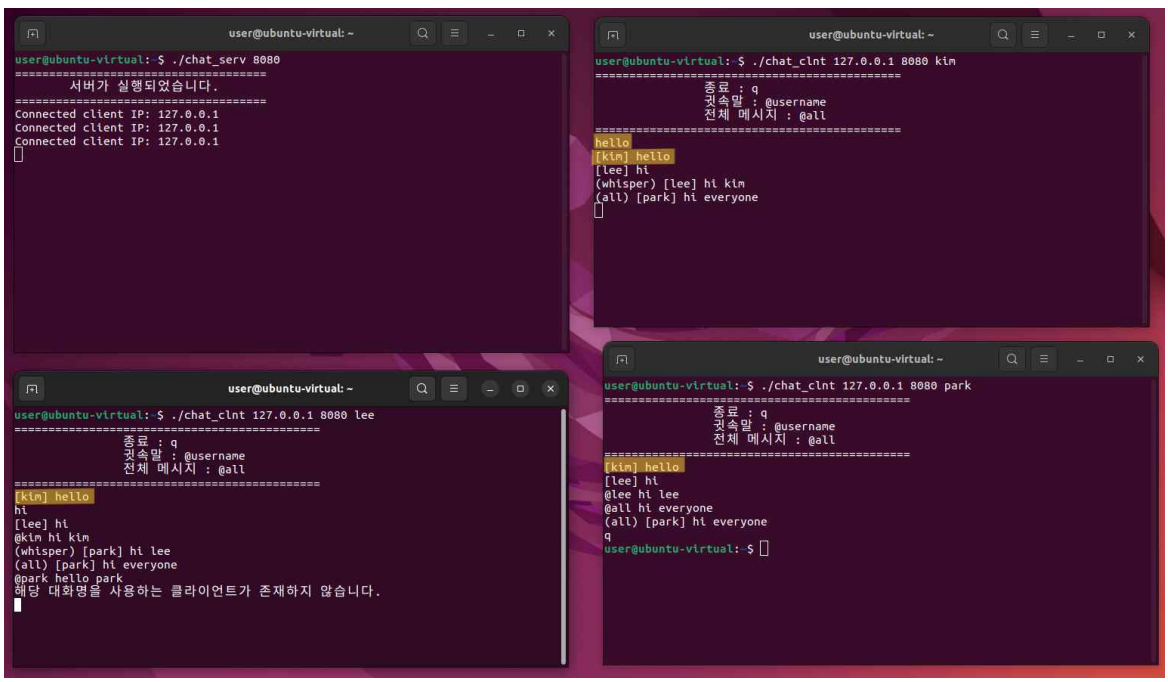
```
        }
        else
                sprintf(name_msg,"%s %s", name, msg);
```

보내려는 메시지가 '@'로 시작할 때 '@receiver [name] text'를 name_msg에 저장한다. 보내려는 메시지가 '@'로 시작하지 않을 때 '[name] text'를 name_msg에 저장한다. 이후 name_msg의 내용을 서버로 전송하고 해당 메소드를 끝낸다.

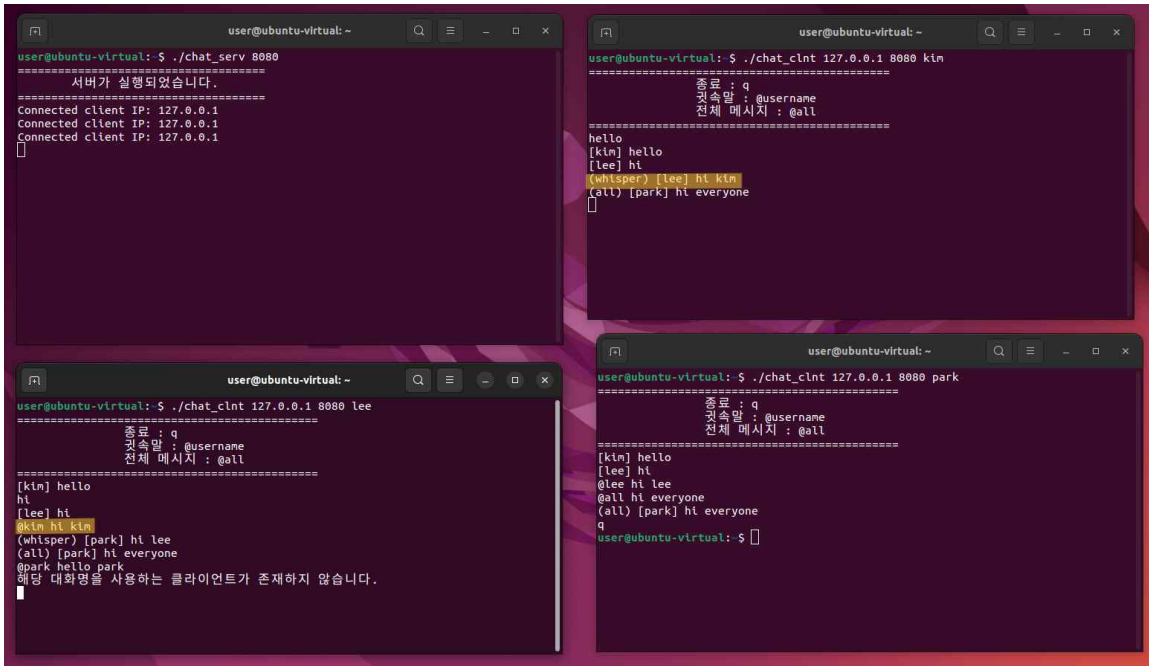
while문 안에서 해당 코드가 반복되어 실행되므로 매 코드의 실행 전 memset을 통해 name_msg를 0으로 초기화한다. 초기화 과정을 넣어야 특정 전송에서 이전의 메시지가 남아 함께 전송되는 현상을 막을 수 있다.

## 3. 실행 결과

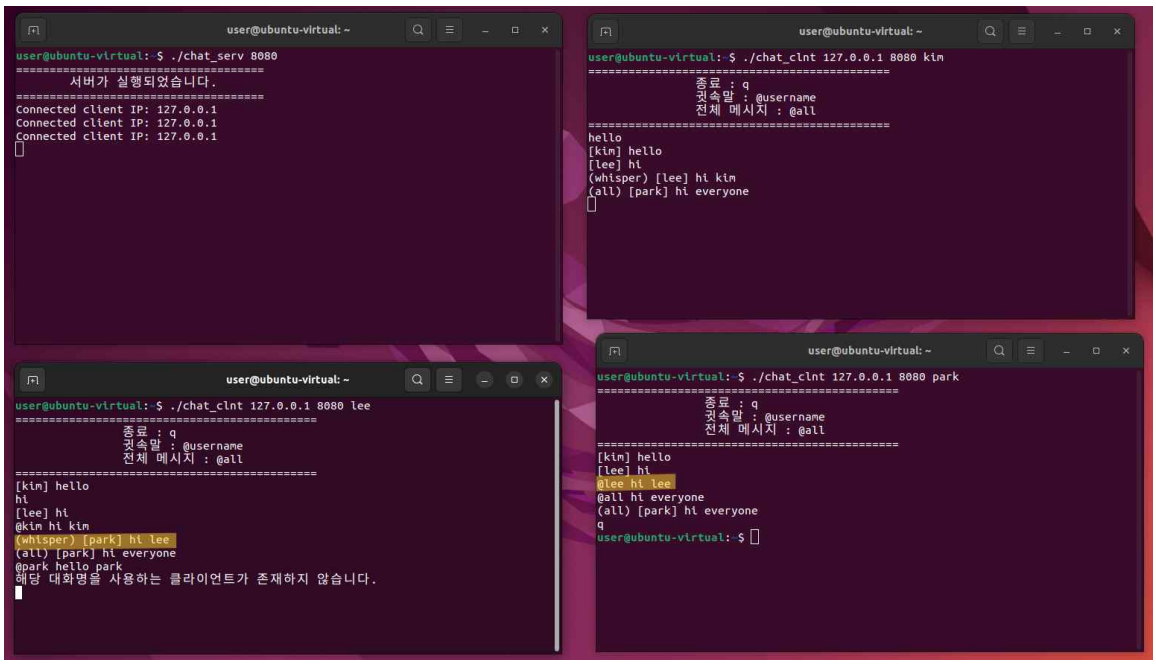
아래 사진들은 위의 1명의 서버와 3명의 클라이언트(kim, lee, park)가 해당 프로그램을 실행한 실행 결과이다. 프로그램의 실행 결과를 단계적으로 살펴보겠다.



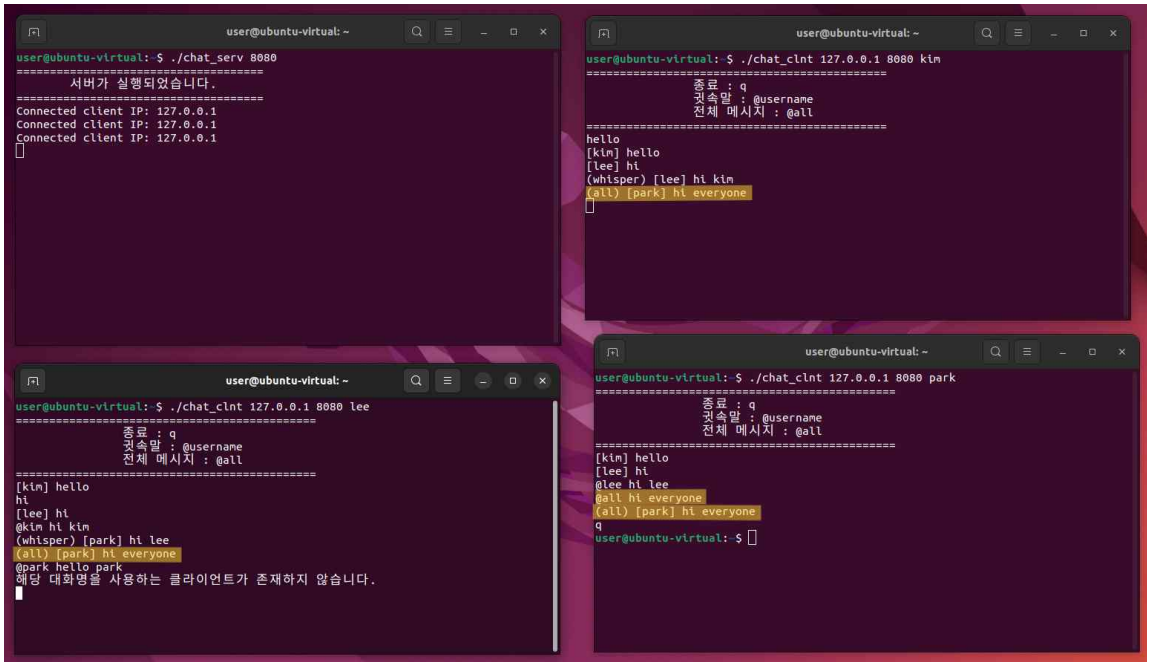

위 사진에서 kim이 보낸 hello라는 메시지가 '[kim] hello'로 모든 클라이언트들에게 전송된 것을 확인할 수 있다.



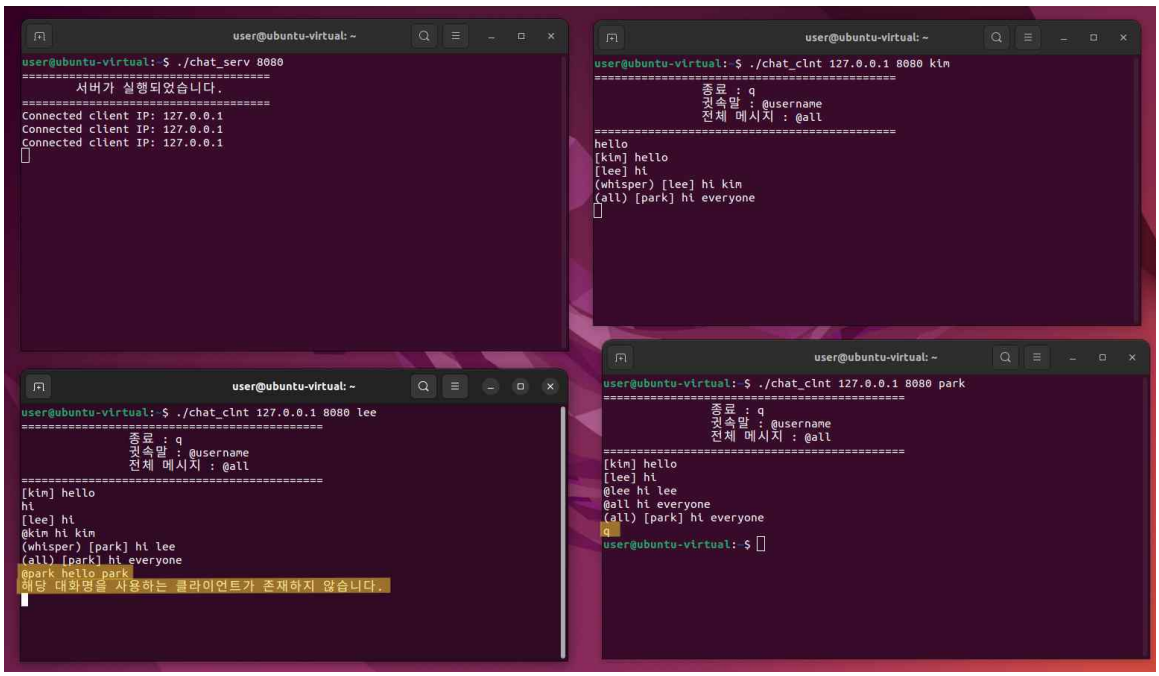

lee가 kim에게 보낸 hi kim이라는 메시지가 kim에게만 '(whisper) [lee] hi kim'으로 전달된 것을 확인할 수 있다.




마찬가지로 park이 lee에게 보낸 메시지가 kim에게는 가지 않고 lee에게만 전송된 것을 확인할 수 있다.




park이 @all 표식을 붙여 전송한 메시지는 (all)이라는 키워드가 붙어 모든 사용자에게 전달되었다.




park이 q를 눌러 프로그램을 종료한 이후, lee가 park에게 1:1 메시지를 보냈지만 '해당 대화명을 사용하는 클라이언트가 존재하지 않습니다.'라는 메시지와 함께 전송에 실패한 것을 확인할 수 있다.