

프로젝트 결과보고서

기본 사항			
교과목명	오픈소스 프로젝트	담당교수	김성우
프로젝트 주제	리눅스 환경의 GUI 기반 채팅 및 파일 전송 메신저		
팀구성원 (학번/성명)	20212977 이규찬, 20212979 임진호, 20233065 이지민		
개발 환경	리눅스 및 GNU 프로그래밍 개발 도구 (gcc, make, gdb) 등		
프로젝트 주소	https://github.com/leekyuchan0215/OpenSource_Project2.git		

1. 프로젝트 개요

본 프로젝트는 리눅스 환경에서 TCP/IP 소켓 프로그래밍과 GTK+ 3.0 라이브러리를 활용하여, 다중 사용자 간의 실시간 채팅 및 파일 전송이 가능한 GUI 메신저 프로그램을 구현하는 것을 목표로 한다. 기존의 CLI 방식의 한계를 넘어, 사용자 친화적인 그래픽 인터페이스(GUI)를 제공하며, 멀티스레드 기법을 도입하여 다수의 클라이언트가 동시에 접속하더라도 끊김 없는 메시지 송수신을 보장한다. 또한, FTP 프로토콜의 개념을 차용하여 텍스트뿐만 아니라 바이너리 데이터(이미지, 문서 등)를 안정적으로 전송하는 기능을 구현함으로써, 운영체제의 시스템 프로그래밍 능력과 네트워크 응용 능력을 종합적으로 배양하고자 한다.

2. 설계 구성 요소

구분	상세 내용
목표 설정	<ul style="list-style-type: none">최종 목표: 리눅스 시스템 함수와 GTK 라이브러리를 결합하여 실용적인 GUI 메신저 개발.팀 목표: 3인의 팀원이 서버, 네트워크 로직, UI 디자인으로 역할을 분담하여 협업 능력을 향상하고, GitHub를 통한 형상 관리를 실습한다.
합성	<ul style="list-style-type: none">기술 융합: 기존의 C언어 소켓 프로그래밍 기술에 오픈소스인 GTK+ 3.0 라이브러리를 합성하여 새로운 GUI 응용 소프트웨어를 제작함.구조적 설계: UI 모듈과 Network 모듈을 분리하고, 이를 Main에서 통합하는 모듈화 설계를 통해 유지보수성과 재사용성을 높임.
분석	<ul style="list-style-type: none">요구사항 분석: 설계지침서의 최소 코드 분량(1,000라인) 충족과 파일 전송 기능 구현을 위해 FTP 프로토콜 및 TCP 패킷 구조를 분석함.기능 분해: 프로젝트를 '접속(Login)', '채팅(Chat)', '파일 전송(File Transfer)'의 3 단계 프로세스로 분해하여 작업 구조(WBS)를 수립함.
제작	<ul style="list-style-type: none">구현: gcc와 make 유ти리티를 사용하여 빌드 자동화 환경을 구축하고, 정해진 일정(11주~14주) 내에 코딩 및 디버깅을 완료함.

	<ul style="list-style-type: none"> 협업: Git을 활용하여 팀원 간 코드를 병합하고 충돌을 해결하며 소프트웨어를 완성함.
시험	<ul style="list-style-type: none"> 기능 테스트: 로컬 루프백(127.0.0.1) 및 내부 네트워크 환경에서 다중 클라이언트 접속 테스트 수행. 안정성 테스트: 대용량 파일 전송 시 데이터 손실 여부와 한글 메시지(UTF-8) 깨짐 현상을 검증하고 수정함.

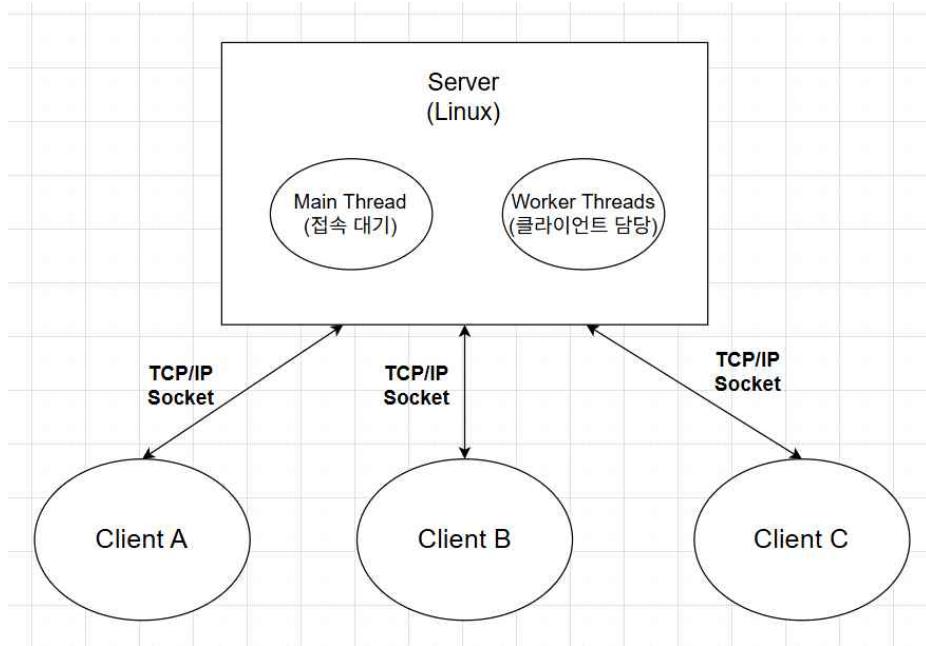
3. 설계 제한 요소

구분	적용 내용 및 준수 사항
경제성	<ul style="list-style-type: none"> 오픈소스 활용: 무료 오픈소스인 Linux(Ubuntu), GCC, GTK+ 라이브러리를 사용하여 라이선스 비용 없이 개발함. 코드 분량 준수: 설계지침서의 요구사항을 준수하기 위해, server.c, client.c를 network.c, ui.c, common.h 등으로 모듈화하고 상세한 주석을 작성하여 구현함.
안정성	<ul style="list-style-type: none"> 멀티스레딩: 서버와 클라이언트 모두 송신과 수신 스레드를 분리하여, 파일 전송 중에도 채팅이 멈추지 않는 비동기 처리를 구현함. 동기화: 다중 클라이언트 접속 시 발생할 수 있는 경쟁 상태를 방지하기 위해 Mutex를 사용하여 임계 구역을 보호함.
생산성	<ul style="list-style-type: none"> 모듈화: UI 코드와 네트워크 통신 코드를 파일별로 분리하여, 추후 기능을 확장하거나 다른 프로젝트에 네트워크 모듈만 재사용할 수 있도록 생산성을 고려함. 이식성: 표준 C 라이브러리와 크로스 플랫폼을 지원하는 GTK를 사용하여 리눅스 환경뿐만 아니라 유닉스 계열 시스템에서의 호환성을 확보함.
신뢰성	<ul style="list-style-type: none"> 프로토콜 정의: 파일 전송 시 [FILE]:파일명:크기와 같은 자체 헤더 프로토콜을 정의하여, 데이터가 섞이거나 유실되지 않고 정확하게 상대방에게 전달되도록 신뢰성을 확보함.
윤리성	<ul style="list-style-type: none"> 라이선스 준수: 프로젝트 결과물에 MIT License를 명시하여, 오픈소스 소프트웨어의 저작권을 준수하고 자유로운 사용권을 보장함.

4. 설계 내용

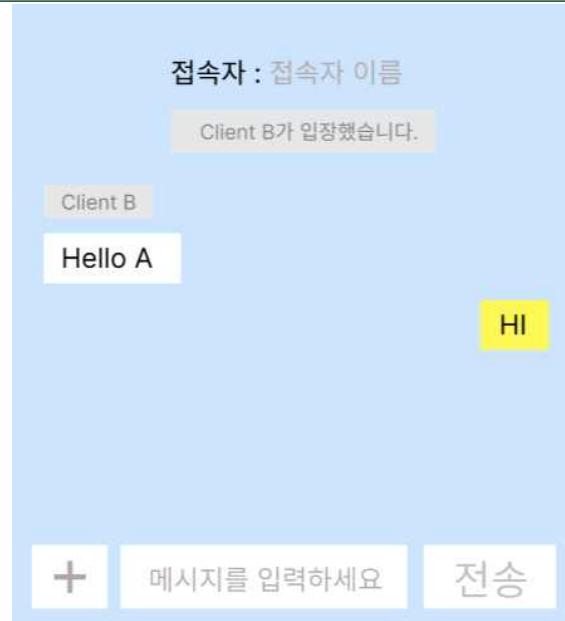
4.1 시스템 아키텍처

본 프로젝트는 리눅스 환경에서 클라이언트-서버 모델(C/S Model)을 기반으로 동작합니다. TCP/IP 소켓을 통해 연결되며, 서버는 멀티스레드(Pthread) 방식을 사용하여 다수의 클라이언트를 동시에 처리합니다.



4.2 GUI 인터페이스 설계

GTK+를 사용하여 채팅창, 파일 전송 버튼 등을 구성



4.3 통신 프로토콜 설계

- 본 프로젝트는 텍스트 메시지와 바이너리 파일 데이터를 하나의 소켓 채널에서 효율적으로 처리하기 위해, 문자열 기반의 애플리케이션 계층 프로토콜을 자체 설계하였습니다.
- 데이터 송수신 시 발생할 수 있는 패킷 유실이나 끊침 현상을 방지하고, 데이터의 유형(채팅/파일)을 명확히 구분하기 위해 다음과 같은 헤더 파싱 방식을 적용하였습니다.

1. 채팅 메시지 패킷



2. 파일 전송 헤더 패킷

Tag	File Name	File Size
[FILE]	project.pdf	2048

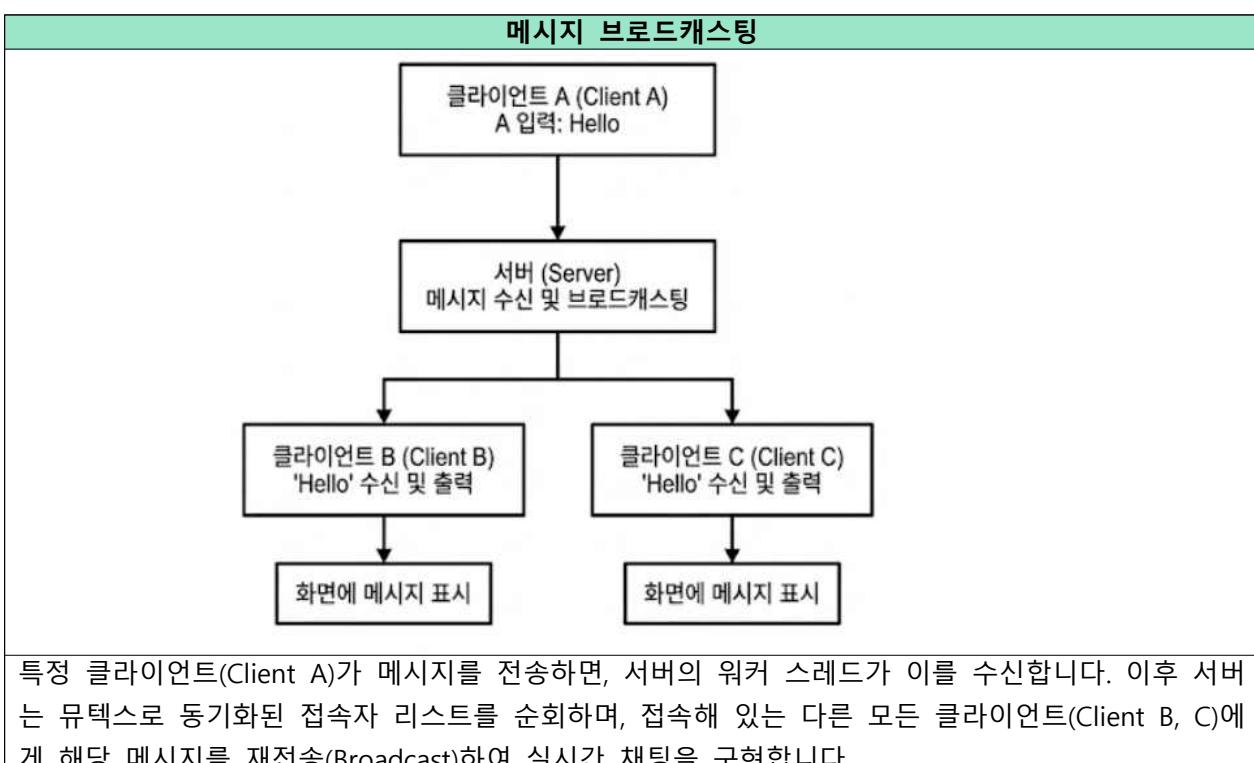
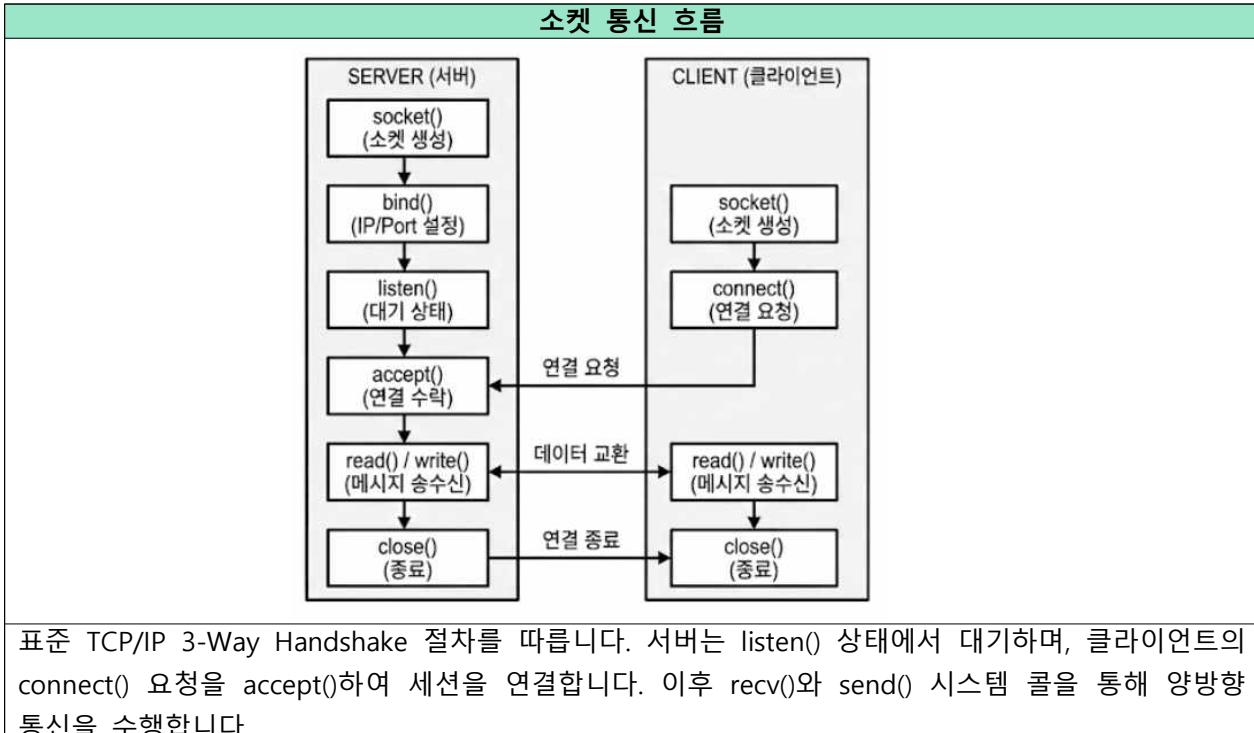
1. 패킷 구조 및 구분

- 구분자 기반 파싱** : 모든 제어 메시지는 콜론(:)을 구분자로 사용하여 필드를 나눕니다. 수신 측에서는 문자열 토큰화를 통해 헤더 정보를 추출합니다.
- 채팅 메시지** : 별도의 헤더 없이 전송되는 일반 문자열은 채팅 메시지로 간주하여, 수신 즉시 채팅창에 출력합니다.
- 파일 전송 헤더** : 파일 전송 시에는 데이터 스트림의 혼선을 막기 위해 [FILE]이라는 고유 태그를 접두어로 사용합니다.

2. 파일 전송 메커니즘

- 헤더 전송 단계** : 파일 전송 요청 시, 먼저 [FILE]:파일명:파일크기 형식의 메타데이터를 전송합니다. 이를 통해 수신 측은 파일 저장을 위한 버퍼를 미리 할당하고 바이너리 수신 모드로 전환합니다.
- 데이터 전송 단계** : 헤더 전송 직후, 실제 파일의 바이너리 데이터를 1024바이트 단위로 분할 전송하여 대용량 파일도 안정적으로 수신할 수 있도록 구현하였습니다.

4.4 시스템 동작 흐름

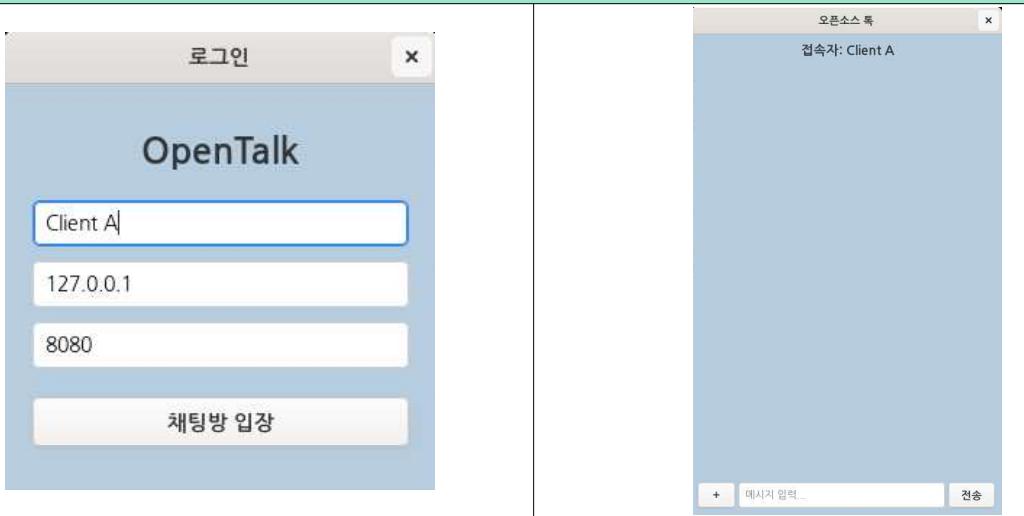


5. 프로젝트 결과

5.1 프로젝트 구현 내용

5.1.1 GUI 구현

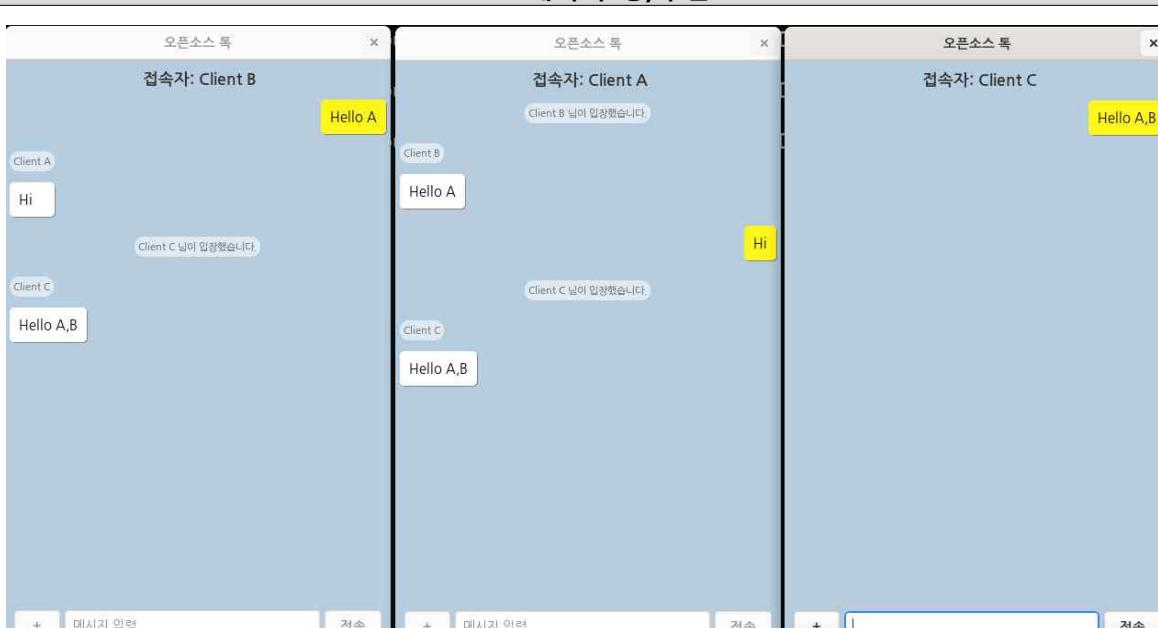
1. 클라이언트 로그인 및 접속 설정 화면



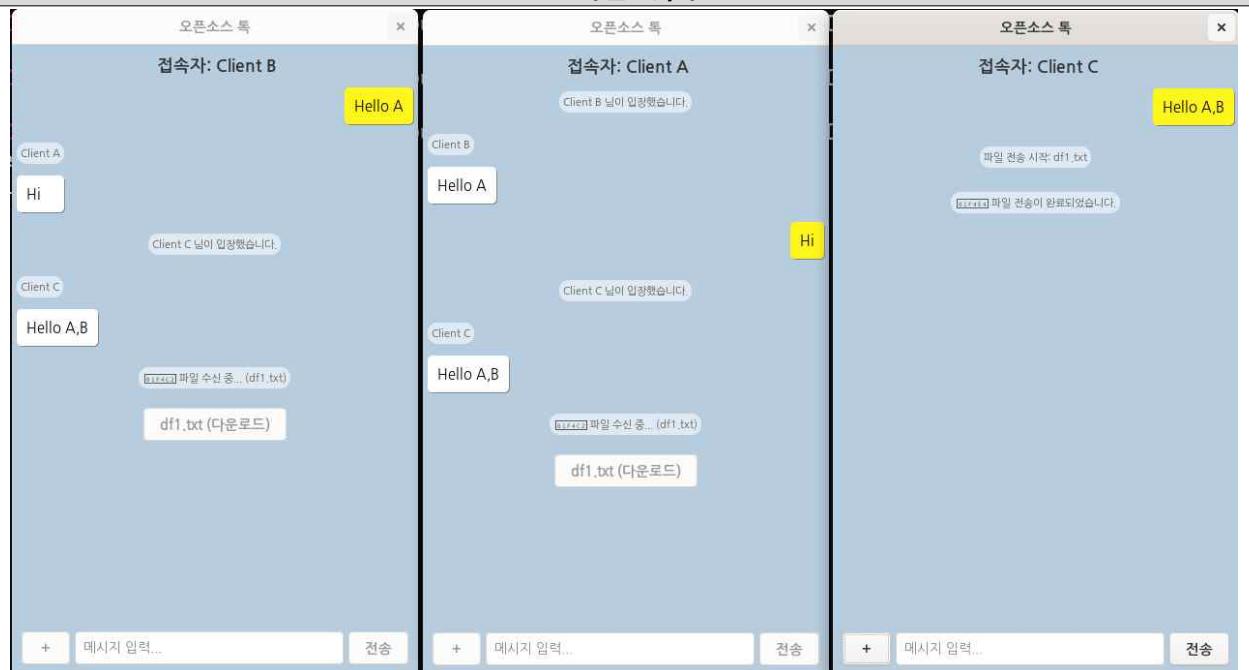
- 접속 정보 설정:** 로컬호스트(127.0.0.1) 및 외부 서버 IP 접속을 지원하며, 서버에 설정된 포트(Port) 번호를 입력하여 소켓 연결을 시도합니다.
- 사용자 식별:** 채팅방 내에서 사용할 닉네임(Client A)을 입력받아, 서버 접속 시 해당 이름으로 세션이 생성됩니다.
- 유효성 검사:** '채팅방 입장' 버튼 클릭 시 입력된 IP와 포트로 connect()를 요청하며, 연결 실패 시 접속이 거부됩니다.

2. 실시간 멀티 채팅 및 동기화

2-1. 메시지 송/수신



2-2. 파일 송/수신



메시지 시각화 (UI/UX) :

- 나의 메시지 : 우측에 노란색 말풍선으로 정렬하여 보낸 메시지임을 명확히 구분하였습니다.
- 상대방 메시지 : 좌측에 흰색 말풍선으로 정렬하며, 메시지 위에 보낸 사람의 닉네임을 표시하여 발신자를 식별할 수 있습니다.

시스템 알림 : 새로운 사용자(Client B, Client C)가 입장할 때마다, 모든 접속자의 화면 중앙에 [입장 알림] 메시지가 실시간으로 브로드캐스팅되는 것을 확인할 수 있습니다.

파일 전송 및 수신 :

- 텍스트뿐만 아니라 파일 전송 시에도 전용 말풍선이 생성됩니다.
- 우측 화면(Client C)에서 파일을 전송하면, 나머지 클라이언트 화면에 다운로드 버튼이 활성화되며, 전송 시작 및 완료 상태가 시스템 메시지로 출력됩니다.

5.1.2 주요 코드 구현

5.1.2.1 서버 코드

1. 서버 접속 처리 및 멀티스레드 (server.c)

서버(server.c)는 다수의 클라이언트를 관리하기 위해 ClientInfo 구조체 배열을 사용하며, accept()로 연결이 수립될 때마다 빈 슬롯(Index)을 찾아 워커 스레드(Worker Thread)를 생성합니다. 이때 접속자 명단이 꼬이지 않도록 Mutex를 사용하여 동기화를 수행합니다.

// 클라이언트 수락 루프

```
while(1) {
    struct sockaddr_in clnt_addr;
    socklen_t sz = sizeof(clnt_addr);
    // 1. 연결 요청 수락 (Blocking)
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &sz);

    // 2. 임계 구역 진입 (Mutex Lock)
    pthread_mutex_lock(&mutex);
    // 빈 슬롯(Socket이 0인 곳)을 찾아 클라이언트 등록
    for(int i=0; i<MAX_CLIENTS; i++) {
        if(clients[i].socket == 0) {
            clients[i].socket = clnt_sock; // 소켓 저장
            clients[i].address = clnt_addr; // 주소 정보 저장

            // 3. 워커 스레드 생성 (Client 별도 처리)
            pthread_t t;
            int *id = malloc(sizeof(int)); // 클라이언트 ID(인덱스) 동적 할당
            *id = i;
            // handle_client 함수가 별도 스레드에서 실행됨
            pthread_create(&t, NULL, handle_client, id);
            break;
        }
    }
    pthread_mutex_unlock(&mutex); // 임계 구역 해제
}
```

- **pthread_mutex_lock(&mutex):** 여러 클라이언트가 동시에 접속할 때, 전역 배열인 clients[]에 충돌이 발생하지 않도록 상호 배제를 적용했습니다.
- **pthread_create(...):** 연결된 클라이언트마다 handle_client 함수를 수행하는 독립적인 스레드를 생성하여, 메인 스레드는 멈추지 않고 계속해서 새로운 접속을 받을 수 있습니다.

2. 클라이언트 요청 처리 및 연결 종료 (server.c)

handle_client 함수는 각 클라이언트 전담 스레드에서 실행됩니다. 클라이언트가 보내온 메시지를 수신(Read)하여 접속된 다른 모든 사용자에게 전송(Broadcast)하고, 연결이 끊기면 소켓 자원을 정리하여 해당 슬롯을 다시 사용할 수 있도록 만듭니다.

// 스레드에서 실행될 클라이언트 처리 함수

```
void *handle_client(void *arg) {
    int id = *(int *)arg; // 매개변수로 전달받은 클라이언트 인덱스(ID) 확보
    int sock = clients[id].socket;
    char buffer[BUFFER_SIZE];
    char name[MAX_NAME_LEN];
    int len;

    // 1. 닉네임 수신
    len = recv(sock, name, MAX_NAME_LEN, 0);
    if (len &lt;= 0) {
        close(sock);
        clients[id].socket = 0; // 연결 해제 처리
        return NULL;
    }
    name[len] = 0;

    // 구조체에 이름 저장
    strcpy(clients[id].name, name);
    clients[id].join_time = time(NULL);
    LOG("User Connected: %s (Slot: %d)", name, id);
    print_client_list();

    // 2. 입장 알림 브로드캐스트
    sprintf(buffer, "[SYSTEM]:%s 님이 입장했습니다.", name);
    broadcast(buffer, strlen(buffer), sock);

    // 3. 메시지 루프
    while ((len = recv(sock, buffer, BUFFER_SIZE, 0)) &gt; 0) {
        // [파일 전송] 헤더나 데이터는 그대로 통과 (Raw Forwarding)
        if (strncmp(buffer, "[FILE]:", 7) == 0 || strncmp(buffer, "[DATA]", 6) == 0) {
            broadcast(buffer, len, sock);
        }
        // [일반 채팅] 이름 붙여서 전송
        else {
            buffer[len] = 0;
            char msg_full[BUFFER_SIZE + 50];
```

```

        sprintf(msg_full, "%s:%s", name, buffer);
        broadcast(msg_full, strlen(msg_full), sock);
    }
}

// 4. 퇴장 처리
LOG("User Disconnected: %s", clients[id].name);

pthread_mutex_lock(&mutex);
close(clients[id].socket);
clients[id].socket = 0;
memset(clients[id].name, 0, MAX_NAME_LEN);
pthread_mutex_unlock(&mutex);
return NULL;
}

```

- 초기 닉네임 수신** : 연결 수립 직후 recv를 통해 대화명을 먼저 받습니다. 이를 구조체(clients)에 저장하고 [SYSTEM] 메시지를 생성해 입장 사실을 알립니다.
- 패킷 분기 (strncmp)** : 수신된 데이터가 [FILE]: 또는 [DATA] 헤더로 시작하는 경우, 파일 전송으로 간주하여 데이터를 가공 없이 그대로 브로드캐스트(Raw Forwarding)합니다. 일반 채팅은 이름:메시지 형식으로 포맷팅하여 전송합니다.
- 동기화 및 종료** : 클라이언트가 종료하면 mutex를 잠그고 소켓과 닉네임 버퍼(memset)를 안전하게 초기화합니다.

3. 메시지 브로드캐스트 및 동기화 (server.c)

한 클라이언트로부터 수신된 메시지(채팅 또는 파일 데이터)를 접속해 있는 다른 모든 클라이언트에게 전송하는 함수입니다. 여러 스레드가 동시에 소켓에 쓰기 작업을 시도할 수 있으므로, Mutex를 사용하여 스레드 안전성(Thread-safety)을 보장해야 합니다.

// 모든 클라이언트에게 메시지 전송

```

void broadcast(char *data, int len, int sender_sock) {
    // 1. 임계 구역 진입 (동기화)
    pthread_mutex_lock(&mutex);
    for (int i = 0; i &lt; MAX_CLIENTS; i++) {
        // 접속된 클라이언트(socket != 0) 중,
        // 메시지를 보낸 당사자(s_sock)를 제외하고 전송
        if (clients[i].socket != 0 && clients[i].socket != sender_sock) {
            // send 함수의 반환값을 확인하여 안정성 확보
            if (send(clients[i].socket, data, len, 0) &lt; 0) {
                ERR("Failed to send message to client %d", clients[i].socket);
            }
        }
    }
}

```

```

        }
    }

// 2. 임계 구역 해제
pthread_mutex_unlock(&mutex);
}

...

//현재 접속자 목록 출력 헬퍼 함수
void print_client_list() {
    printf("===== Current Users =====\n");
    for (int i = 0; i & MAX_CLIENTS; i++) {
        if (clients[i].socket != 0) {
            printf("[%d] Name: %s (IP: %s)\n",
                   i,
                   clients[i].name,
                   inet_ntoa(clients[i].address.sin_addr));
        }
    }
    printf("===== \n");
}

```

- **pthread_mutex_lock/unlock:** 여러 명의 클라이언트가 동시에 채팅을 입력할 때, 메시지가 섞이거나 소켓 접근이 충돌하는 것을 방지하기 위해 전송 로직 전체를 임계 구역으로 설정했습니다.
- **s_sock 체크:** clients[i].socket != s_sock 조건을 추가하여, 메시지를 보낸 본인에게는 데이터가 다시 전송(Echo)되지 않도록 처리했습니다. (클라이언트 UI에서 본인의 메시지는 스스로 출력한다고 가정)

5.1.2.2 클라이언트 통신 구현

1. 서버 연결 요청 및 초기화

클라이언트는 소켓을 생성하여 서버의 IP와 포트로 TCP 연결을 시도합니다. 연결이 수립(Success)되면, 즉시 전역 변수에 저장된 사용자의 닉네임(my_name)을 서버로 전송하여 접속자를 등록하는 절차를 수행합니다.

```
// 서버 연결 및 수신 스레드 시작 함수
int connect_to_server(const char *ip, int port) {
    struct sockaddr_in serv_addr;

    // 1. 소켓 생성
    sock = socket(PF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Socket Creation Failed");
        return -1;
    }
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(ip);
    serv_addr.sin_port = htons(port);

    // 2. 연결 시도
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1) {
        perror("Connection Failed");
        return -1;
    }

    // 3. 접속 성공 시 닉네임 전송
    if (send(sock, my_name, strlen(my_name), 0) & 0) {
        perror("Send Name Failed");
        return -1;
    }

    LOG("Connected to Server %s:%d", ip, port);
    return 0;
}
```

- **socket(PF_INET, SOCK_STREAM, 0):** TCP/IP 통신을 위한 스트림 소켓을 생성합니다.
- **connect(...):** struct sockaddr_in 구조체에 설정된 서버의 주소 정보를 바탕으로 연결 요청(3-way Handshake)을 보냅니다.
- **send(..., my_name, ...):** 연결이 성공하자마자 가장 먼저 닉네임 패킷을 보냅니다. 이는 서버의 handle_client에서 recv로 대기하고 있는 부분과 짹을 이뤄, 서버가 클라이언트를 식별할 수 있게 합니다.

2. 메시지 수신 및 데이터 파싱 스레드

서버와 연결된 이후, 클라이언트는 언제 도착할지 모르는 데이터를 대기해야 합니다. 메인 UI(GTK Main Loop)가 멈추지 않도록 별도의 스레드(recv_msg_thread)에서 수신을 담당하며, 수신된 패킷의 헤더([FILE]:, [DATA])를 분석하여 파일 다운로드와 일반 채팅을 구분하여 처리합니다.

```
void* recv_msg_thread(void *arg) {
    char buf[BUFFER_SIZE];
    int len;
    while ((len = recv(sock, buf, BUFFER_SIZE, 0)) & 0) {

        // =====
        // CASE 1: 파일 데이터 수신 중
        // =====
        if (is_receiving_file) {
            if (strncmp(buf, "[DATA]", 6) == 0) {
                int data_len = len - 6;
                // 남은 크기보다 더 많이 들어오면 자름 (안전장치)
                if (data_len & recv_remain_size) data_len = recv_remain_size;

                fwrite(buf + 6, 1, data_len, recv_fp);
                recv_remain_size -= data_len;
                // 다운로드 완료 체크
                if (recv_remain_size &= 0) {
                    fclose(recv_fp);
                    recv_fp = NULL;
                    is_receiving_file = 0;

                    LOG("File Download Finished: %s", recv_filename);

                    // UI에 다운로드 버튼 생성 요청
                    add_file_download_btn(recv_filename);
                }
            }
            continue;
        }
        // =====
        // CASE 2: 새로운 파일 전송 시작 헤더 감지
        // =====
        if (strncmp(buf, "[FILE]:", 7) == 0) {
            char *ptr = buf + 7;
```

```

char *size_ptr = strchr(ptr, ':');

if (size_ptr) {
    *size_ptr = 'W0';
    strcpy(recv_filename, ptr);
    recv_remain_size = atol(size_ptr + 1);

    // 임시 파일 생성
    sprintf(temp_filepath, "temp_%s", recv_filename);
    recv_fp = fopen(temp_filepath, "wb");

    if (recv_fp) {
        is_receiving_file = 1;
        char alert[256];
        sprintf(alert, "📁 파일 수신 중... (%s)", recv_filename);
        add_system_msg(alert);
    } else {
        ERR("Failed to create temp file");
    }
}
continue;
}

// =====
// CASE 3: 일반 텍스트 채팅
// =====
buf[len] = 0;
char *utf8 = convert_to_utf8(buf);
char *sep = strchr(utf8, ':');

if (sep) {
    *sep = 'W0'; // 이름과 메시지 분리
    // 상대방 메시지로 UI에 추가
    add_chat_bubble(utf8, sep + 1, 0);
} else {
    // 시스템 메시지 처리
    if (strstr(utf8, "[SYSTEM]")) {
        char *sys_sep = strchr(utf8, ':');
        if (sys_sep) add_system_msg(sys_sep + 1);
    }
}

```

```

    }

    g_free(utf8);
}

// 연결 종료 시
LOG("Disconnected from server");
add_system_msg("X 서버와의 연결이 끊어졌습니다.");
return NULL;
}

```

- **상태 기반 분기 (is_receiving_file):** [FILE]: 헤더를 받으면 파일 수신 모드(1)로 전환되고, 이후 들어오는 [DATA] 패킷들은 텍스트로 처리하지 않고 즉시 파일(fwrite)에 씁니다.
- **프로토콜 파싱 (strncmp):** 서버에서 정의한 규칙대로 [FILE]:파일명:크기 헤더를 분석하여 파일명과 전송받을 바이트 수를 추출합니다.
- **UI 스레드 연동:** 네트워크 스레드에서 직접 UI를 수정하면 충돌이 발생할 수 있으므로, add_chat_bubble이나 add_file_download_btn과 같은 헬퍼 함수를 통해 안전하게 UI 업데이트를 요청합니다.
- **인코딩 변환:** convert_to_utf8 함수를 사용하여 한글이 깨지지 않도록 인코딩을 맞춘 뒤 출력합니다.

3. 메시지 및 파일 전송 구현

일반 텍스트 메시지는 send_text_message 함수를 통해 즉시 전송되지만, 대용량 파일의 경우 전송 시간이 길어질 수 있어 UI가 멈추는(Freezing) 현상을 방지하기 위해 별도의 스레드(send_file_thread)에서 비동기적으로 처리합니다. 파일 전송 프로토콜은 헤더(메타데이터)와 데이터(내용)로 구분하여 전송합니다.

```

/**
 * @brief 텍스트 메시지를 서버로 전송합니다.
 * @param msg 보낼 메시지 내용
 */
void send_text_message(const char *msg) {
    if (sock & 0) return;
    if (send(sock, msg, strlen(msg), 0) & 0) {
        perror("Message Send Failed");
    }
}

/**
 * @brief 파일을 서버로 전송하는 스레드 함수
 * @details 대용량 파일 전송 시 UI 블로킹을 막기 위해 별도 스레드에서 실행됩니다.

```

```

/*
void* send_file_thread(void *arg) {
    char *filename = (char *)arg;

    // 파일 열기
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        ERR("Cannot open file: %s", filename);
        g_free(filename);
        return NULL;
    }

    // 파일 크기 측정
    fseek(fp, 0, SEEK_END);
    long filesize = ftell(fp);
    rewind(fp);
    LOG("Start File Upload: %s (%ld bytes)", filename, filesize);

    // 1. 헤더 전송 [FILE]:파일명:크기
    char header[512];
    sprintf(header, "[FILE]:%s:%ld", g_path_get_basename(filename), filesize);
    send(sock, header, strlen(header), 0);

    // 서버 버퍼링 대기
    usleep(50000);

    // 2. 데이터 청크 전송
    char buffer[BUFFER_SIZE];
    char send_buf[BUFFER_SIZE + 10];
    size_t read_size;
    long total_sent = 0;
    while ((read_size = fread(buffer, 1, BUFFER_SIZE - 6, fp)) & 0) {
        memcpy(send_buf, "[DATA]", 6);
        memcpy(send_buf + 6, buffer, read_size);

        if (send(sock, send_buf, read_size + 6, 0) & 0) {
            perror("File Data Send Error");
            break;
        }
        total_sent += read_size;
        usleep(1000); // 네트워크 혼잡 방지
    }
}

```

```

LOG("File Upload Completed: %ld / %ld bytes", total_sent, filesize);
fclose(fp);
g_free(filename);

add_system_msg("업로드 파일 전송이 완료되었습니다.");
return NULL;
}

```

- 스레드 기반 파일 전송:** 파일 전송 중에도 채팅이나 다른 UI 조작이 가능하도록 pthread_create를 통해 생성된 스레드 내에서 파일을 읽고 전송합니다.
- 헤더와 데이터 분리:** 수신 측(서버 및 다른 클라이언트)에서 파일 전송이 시작됨을 알 수 있도록 [FILE]:파일명:파일크기 형식의 헤더를 먼저 보낸 후, 실제 내용은 [DATA] 접두어를 붙여 패킷 단위로 나누어 보냅니다.
- usleep(1000):** 너무 빠른 속도로 패킷을 연속 전송하면 네트워크 버퍼가 넘치거나(Server Overflow) 데이터가 유실될 수 있어, 패킷 사이사이에 미세한 지연 시간을 두어 안정성을 확보했습니다.

5.1.3 실행

1. make

```

rbcks@DESKTOP-09DVRUV:~/OpenSource_Project2$ make
gcc server.c -o server -I/usr/include/gtk-3.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/webp -I/usr/include/gio-unix-2.0 -I/usr/include/atk-1.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -pthread -lgtk-3 -lgdk-3 -lz -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgbt-2.0 -lglib-2.0 -lpthread
gcc main.c network.c ui.c -o client -I/usr/include/gtk-3.0 -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/harfbuzz -I/usr/include/freetype2 -I/usr/include/libpng16 -I/usr/include/libmount -I/usr/include/blkid -I/usr/include/fribidi -I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/gdk-pixbuf-2.0 -I/usr/include/x86_64-linux-gnu -I/usr/include/webp -I/usr/include/gio-unix-2.0 -I/usr/include/atk-1.0 -I/usr/include/at-spi2-atk/2.0 -I/usr/include/at-spi-2.0 -I/usr/include/dbus-1.0 -I/usr/lib/x86_64-linux-gnu/dbus-1.0/include -pthread -lgtk-3 -lgdk-3 -lz -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgbt-2.0 -lglib-2.0 -lpthread
network.c: In function 'recv_msg_thread':
network.c:201:57: warning: '%s' directive writing up to 255 bytes into a region of size 229 [-Wformat-overflow=]
    201 |         sprintf(alert, "파일 수신 중... (%s)", recv_filename);
                  ^~~~~~
network.c:201:21: note: 'sprintf' output between 29 and 284 bytes into a destination of size 256
    201 |         sprintf(alert, "파일 수신 중... (%s)", recv_filename);
                  ^~~~~~

```

복잡한 gcc 명령어 대신하여 실행 파일 2개 생성

1. 서버 제작 (server 파일 생성)

- server.c를 컴파일합니다.
- 이때 멀티스레드(-lpthread) 설정도 알아서 다 붙여줍니다.

2. 클라이언트 제작 (client 파일 생성)

- main.c, network.c 등 흩어진 파일들을 하나로 합칩니다.
- 복잡한 GTK 라이브러리 경로(pkg-config ...)도 알아서 다 연결해줍니다.

결론: make 명령어로 서버와 클라이언트 프로그램이 바로 실행 가능한 상태로 완성됩니다.

2. server

```
rbcks@DESKTOP-09DVRUV:~/OpenSource_Project2$ ./server 8080
>> Server Started on Port 8080
>> Waiting for connections...
```

1. **./server**: 만들어진 서버 프로그램을 실행시킵니다.
2. **8080**: 들어오는 문 번호(포트)를 8080번으로 설정합니다.

3. client

```
rbcks@DESKTOP-09DVRUV:~/OpenSource_Project2$ ./client
(client:2888): GLib-GIO-CRITICAL **: 21:57:02.762: g_dbus_proxy_new: assertion 'G_IS_DBUS_CONNECTION (connection)' failed
(client:2888): GLib-GIO-CRITICAL **: 21:57:02.762: g_dbus_proxy_new: assertion 'G_IS_DBUS_CONNECTION (connection)' failed
(client:2888): GLib-GIO-CRITICAL **: 21:57:02.762: g_dbus_proxy_new: assertion 'G_IS_DBUS_CONNECTION (connection)' failed
```



1. **GUI 화면 출력**: GTK 라이브러리를 통해 채팅창 윈도우를 모니터에 띄우고, 사용자 입력을 받을 준비를 합니다.
2. **서버 연결 요청**: 동시에 내부적으로 connect() 함수를 실행하여, 사용자가 입력한 IP와 포트 번호의 서버로 접속을 시도합니다.
3. **수신 대기 모드 진입**: 연결이 성공하면 즉시 수신 전용 스레드를 생성하여, 채팅이나 파일이 들어오는지 실시간으로 감시하기 시작합니다.

5.2 프로젝트 완성도 평가

프로젝트 기능	완성도	설명
소켓 서버 구현	100%	Mutex와 멀티스레드(pthread)를 활용하여 다수의 클라이언트 접속을 안정적으로 처리하는 서버 구축 완료
GUI 인터페이스	100%	GTK+ 라이브러리를 사용하여 클라이언트 접속 화면, 채팅창, 파일 전송 버튼 등 UI 레이아웃 구현 완료
채팅 송수신 기능	80%	서버를 통한 메시지 브로드캐스팅 로직은 정상 작동하나, 한글 입력 시 인코딩 호환 문제로 텍스트가 깨지는 현상이 존재함
파일 전송 기능	90%	스레드를 이용한 대용량 파일 바이너리 전송 로직은 성공적으로 구현했으나, 전송 완료 알림 메시지 출력 시 특수문자 깨짐 현상 발생

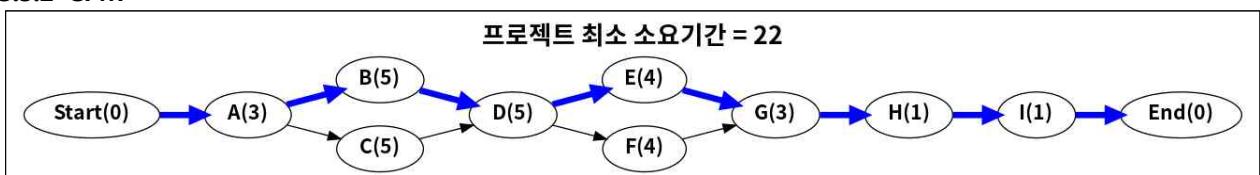
- 소켓 통신 및 로직:** 서버와 클라이언트 간의 연결(Connect), 메시지 전달(Send/Recv), 파일 스트림 처리 등 핵심 네트워크 로직은 정상적으로 구현되었습니다.
- 한계점:** 개발 환경(Linux/Windows)과 GTK 라이브러리 간의 문자셋(UTF-8 vs CP949) 차이로 인해 한글 메시지와 일부 시스템 알림 문자가 정상적으로 표시되지 않는 문제가 확인되었습니다. 이는 추후 iconv 라이브러리 등을 통한 인코딩 변환 작업으로 개선이 필요합니다.

5.3 일정 계획 평가

5.3.1 소작업 목록

Activity No.	소작업	소요기간	선행작업
Start	시작	0	-
A	프로젝트 기획 및 기술분석	3	Start
B	통신 프로토콜 및 패킷 정의	5	A
C	UI/UX 스토리보드 작성	5	A
D	스레드 및 동기화 로직 설계	5	B,C
E	서버 소켓 코어 구현	4	D
F	클라이언트 GUI 및 수신 연동	4	D
G	파일 전송 모듈 구현 및 통합	3	E,F
H	디버깅 및 인코딩 최적화	1	G
I	최종 산출물 작성	1	H
End	종료	0	I

5.3.2 CPM



5.4 역할 수행 평가

이름	담당 역할	주요 수행 내용 및 평가
이규찬	PM & Main	<p>[시스템 통합 및 빌드 관리]</p> <ul style="list-style-type: none"> main 함수 진입점 관리 및 프로그램 실행 인자(argv) 파싱 처리 Server/Client 실행 파일을 생성하는 Makefile 작성 및 빌드 자동화 Network와 UI 모듈 간의 의존성 관리 및 전체 소스코드 통합 <p>• 평가: 분리된 모듈을 성공적으로 결합하여 실행 흐름을 완성함.</p>
임진호	Network	<p>[서버 구축 및 통신 프로토콜 설계]</p> <ul style="list-style-type: none"> socket API를 활용한 TCP/IP 연결 및 멀티스레드 서버 구현 Mutex를 이용한 임계 구역 동기화 처리 파일 전송을 위한 헤더/데이터 패킷 프로토콜 설계 및 구현 <p>• 평가: 다중 클라이언트 접속 및 파일 바이너리 전송 로직을 안정적으로 구현함.</p>
이지민	GUI	<p>[GUI 디자인 및 이벤트 핸들링]</p> <ul style="list-style-type: none"> Glade 툴을 활용한 채팅창, 입력창, 전송 버튼 등 UI 레이아웃 구성 GTK+ 라이브러리(g_signal_connect)를 사용한 버튼 클릭 및 키보드 이벤트 처리 파일 탐색기 파일로그 연동 및 시스템 알림 메시지 출력 구현 <p>• 평가: 사용자가 직관적으로 사용할 수 있는 인터페이스를 완성도 있게 구현함.</p>

5.5 위험 처리

위험 요인	발생 원인	대처 방안 및 결과
한글 메시지 인코딩 깨짐	<ul style="list-style-type: none"> Linux와 Windows 또는 GTK 내부 인코딩 간의 문자셋 불일치 소켓 전송 시 바이트 스트림 처리 과정에서 멀티바이트 문자 손상 	<ul style="list-style-type: none"> [임시 조치] convert_to_utf8 함수를 통해 수신 메시지 변환 시도 [현황] 영문은 정상 작동하나, 한글 입력 시 부분적으로 깨지는 현상 잔존 (향후 libiconv 라이브러리 도입을 통해 완벽한 변환 로직 적용 필요)
파일 전송 알림 문자 오류	<ul style="list-style-type: none"> 전송 완료 메시지 포맷팅 과정에서 널 문자(₩0) 처리 미흡 또는 메모리 더미 값 출력 "파일 전송 완료" 문구 옆에 의미 없는 특수문자가 붙어서 출력됨 	<ul style="list-style-type: none"> [분석] sprintf로 버퍼 병합 시, 버퍼 초기화를 강화하고 문자열 끝처리를 명확히 하여 해결 시도 단순 UI 표기 오류로, 실제 파일 데이터 무결성에는 영향 없음 확인
대용량 전송 시 UI 프리징	<ul style="list-style-type: none"> 메인 스레드(Main Thread)에서 파일 입출력과 소켓 전송을 수행할 경우, 전송이 끝날 때까지 GUI가 멈추는 현상(Blocking) 	<ul style="list-style-type: none"> [해결] send_file_thread를 별도로 생성하여 비동기 방식으로 파일 전송 처리 파일 전송 중에도 채팅 및 창 이동이 가능하도록 구현 완료
다중 접속 시 데이터 충돌	<ul style="list-style-type: none"> 여러 클라이언트가 동시에 메시지를 보낼 때, 서버의 공유 자원(소켓 버퍼)에 경쟁 상태 발생 	<ul style="list-style-type: none"> [해결] 서버 측 pthread_mutex_lock/unlock을 도입하여 임계 구역(Critical Section) 보호 순차적 메시지 처리로 데이터꼬임 방지 성공

6. 소감

이규찬	<p>팀장으로서 가장 힘들었던 건, 각자 맡은 코드를 하나로 합치는 과정이었습니다. 분명 각자 컴퓨터에서는 잘 돌아갔는데, 이걸 main에서 합치기만 하면 에러가 나거나 실행이 안 되는 경우가 많았습니다. 특히 임진호 팀원이 짠 서버 코드와 이지민 팀원이 짠 UI를 연결할 때 변수명이 겹치거나 헤더 파일이 꼬이는 문제를 해결하느라 꽤 고생했습니다.</p> <p>가장 아쉬운 건 역시 '한글 깨짐' 문제입니다. 리눅스랑 윈도우의 문자셋 차이 때문이라는 건 알았지만, 마감 기한 내에 핵심 기능인 파일 전송부터 완성해야 해서 끝내 고치지 못한 게 계속 마음에 걸립니다. 그래도 Makefile로 빌드 환경을 딱 구축해놓고, 터미널에서 채팅이랑 파일이 훙휙 날아가는 걸 봤을 때의 쾌감은 잊지 못할 것 같습니다. "협업은 코딩보다 소통이 더 중요하다"는 걸 느낀 프로젝트였습니다.</p>
임진호	<p>이번 프로젝트에서 저는 클라이언트 즉 네트워크 통신 로직을 담당하며, 서버와의 소켓 연결부터 메시지 수신 처리, 파일 송·수신 상태 관리까지 전반적인 통신 흐름을 구현하였습니다. 하나의 소켓에서 채팅 메시지와 파일 데이터를 동시에 처리해야 했기 때문에, 수신 패킷을 구분하기 위한 간단한 프로토콜을 설계하고 상태 변수(is_receiving_file)를 기반으로 동작을 분기하는 구조를 직접 구현한 점이 가장 인상 깊었습니다. 특히 파일 수신 중에는 일반 채팅 로직이 섞이지 않도록 흐름을 제어하고, 남은 파일 크기를 기준으로 다운로드 완료 시점을 판단하는 과정에서 네트워크 스트림 처리의 어려움을 실감할 수 있었습니다. 또한 수신 스레드와 UI 로직을 분리하여 네트워크 처리가 화면 동작에 영향을 주지 않도록 구성하면서, 멀티스레드 환경에서의 역할 분리가 왜 중요한지 명확히 이해하게 되었습니다. 이번 프로젝트를 통해 TCP 기반 통신 구조와 데이터 흐름을 코드 수준에서 체계적으로 다룰 수 있게 되었고, 시스템 프로그래밍에 대한 자신감을 얻을 수 있었습니다.</p>
이지민	<p>처음엔 "C언어로 카카오톡 같은 걸 어떻게 만들지?" 막막하기만 했습니다. GTK라는 라이브러리를 처음 써봤는데, 버튼 하나 만드는 것도 생각보다 복잡했습니다. 제일 애를 먹었던 건 파일 전송 버튼을 누르면 전송이 끝날 때까지 채팅창이 멈춰버리는 현상이었습니다. 교수님 강의 때 배운 '스레드'가 왜 필요한지 그때 확 와닿았고, 전송 기능을 별도 스레드로 빼서 해결했을 때 성취감을 느꼈습니다.</p> <p>완성된 화면을 보면 한글도 좀 깨지고 투박해 보이지만, 제가 만든 버튼을 눌러서 파일이 전송되고 알림창이 뜨는 걸 보니 신기했습니다. 백엔드에서 아무리 코드를 잘 짜도, 결국 사용자 눈에 보이는 건 UI라는 생각에 책임감을 가지고 작업했습니다. 다음엔 인코딩 문제도 꼭 해결해서 더 완벽한 프로그램을 만들어보고 싶습니다.</p>