PORTFOLIO

TCP/IP 통신을 이용한 3인 총알 피하기 게임

박장호, 임철진, 최지운

CONTACT

박장호

010-4825-9941

35379289p@gmail.com

목차

01 게임 소개

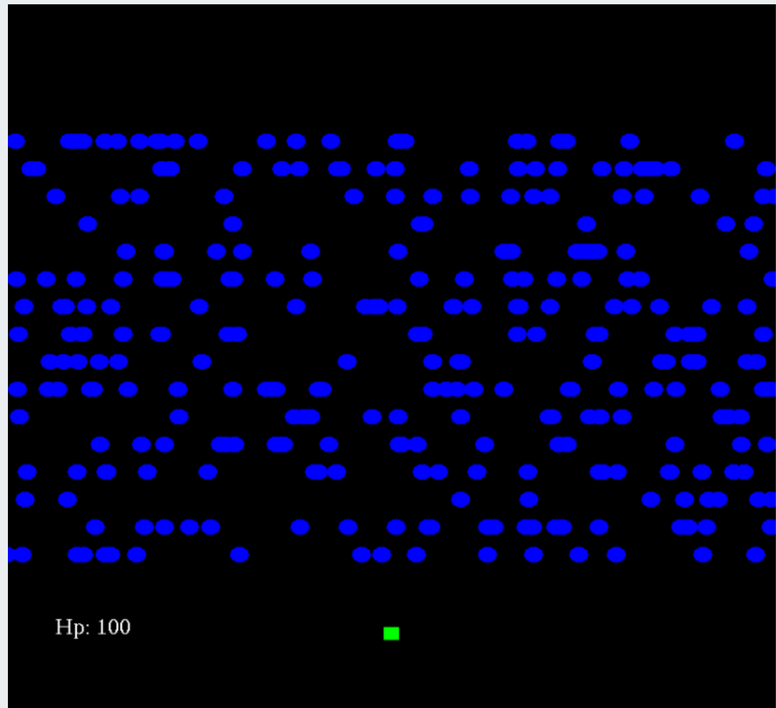
Avoiding Bullets

TCP/IP 프로토콜을 이용한 생존 게임

각 클라이언트들이 서버에 접속 하고 모두 준비 상태가 된다면 게임이 시작됩니다.
게임은 단순하게 하늘에서 날아오는 Bullet들을 피하는 게임이며, 각자의 hp 는 100 으로 bullet과 충돌 시 bullet의 damage(10)만큼 피해를 입습니다.
총 3 명의 플레이어 중 최종적으로 살아남는 플레이어가 승리하게 됩니다.

C C++ OpenGL TCP/IP

GitHub : https://github.com/jangho-park-dev/TCP-IP-Avoiding-bullets
Youtube : https://youtu.be/-iqCcRumO7Y



02 역할 분담

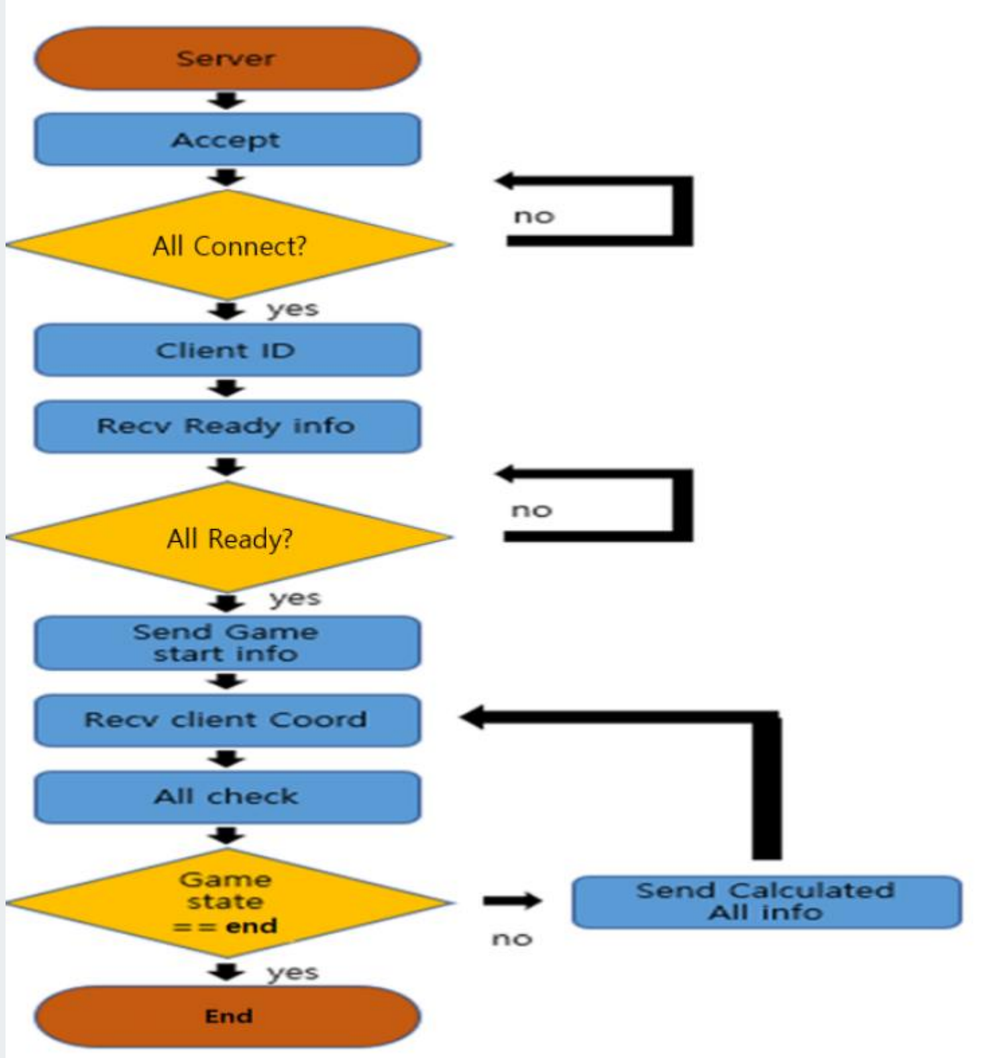
>>> **나의 역할**

- send_Ready()와 recv_Ready()로 ready 정보를 주고받을 수 있도록 했습니다.
- Recv_PosInfo()로 모든 오브젝트의 위치 정보를 받을 수 있도록 했습니다.
- hpUpdate()로 플레이어들의 hp 정보를 업데이트할 수 있도록 했습니다.
- obstacleUpdate()로 총알의 정보를 업데이트할 수 있도록 했습니다.
- recv_AllInfo()와 send_AllInfo(), ProcessClient()는 팀원들과의 지속적인 의사소통으로 만들었습니다.

	세부사항	임철진	박장호	최지운
클라이언트	씬나누기	O		
	send_Ready()		O	
	recv_initInfo()			O
	send_PosInfo()			O
	recv_AllInfo()	O	O	O
	update()	O		
	drawReadyScene	O		
	drawEndScene	O		
서버	recv_Ready()		O	
	send_initInfo()			O
	recv_PosInfo()		O	
	CollisionCheck()	O		
	gameStateUpdate()			O
	hpUpdate()		O	
	obstacleUpdate()		O	
	playerPosUpdate()	O		
	AllReady()			O
	sendAllInfo()	O	O	O
	ProcessClient()	O	O	O

03 구조 소개

>>> High-Level Design

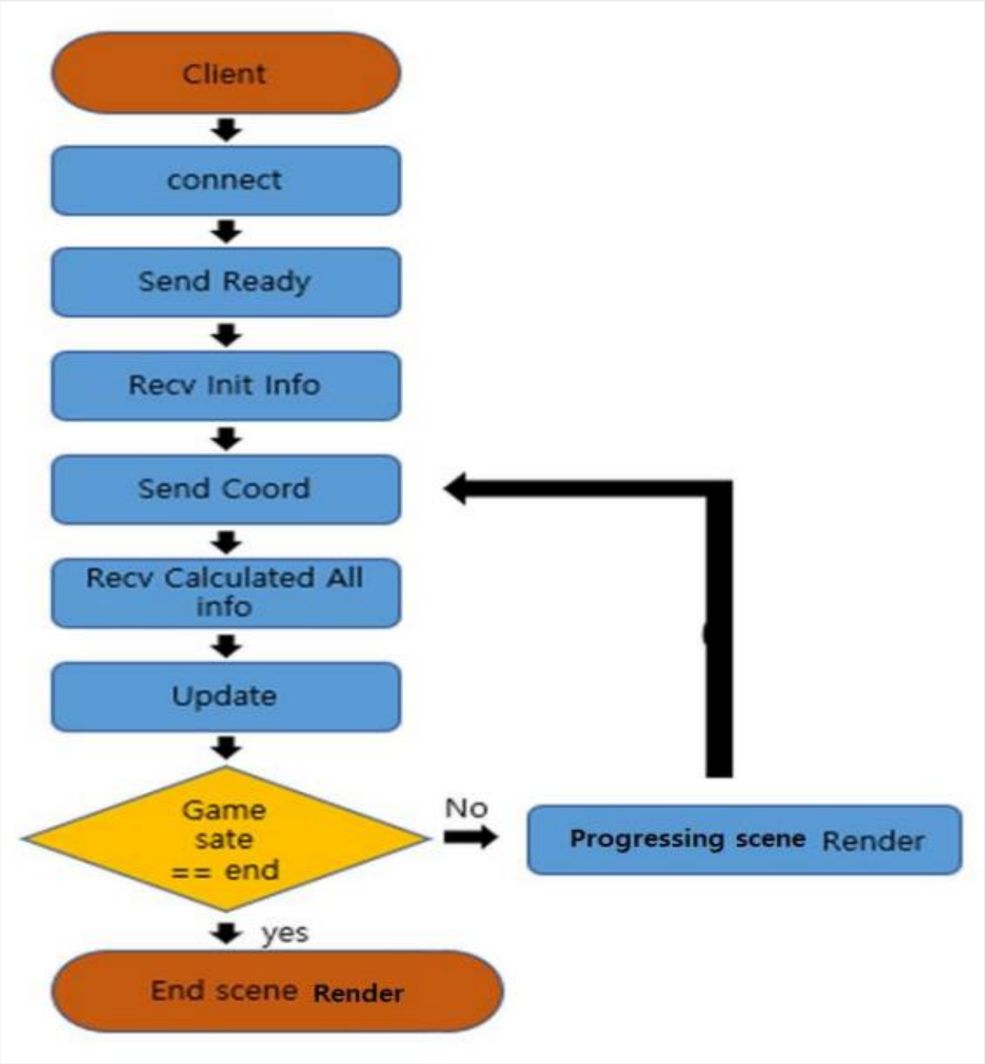


서버

- 서버는 모든 클라이언트가 접속할 때까지 기다립니다.
- 모든 클라이언트에게 접속 정보를 전송 받고, 클라이언트 소켓을 관리하는 배열에 set socket 해줍니다.(accept)
- 각 클라이언트로부터 ready 정보를 전송 받습니다.
- 모두 준비됐다면 게임이 시작되고 게임 시작 초기 정보(초기 위치, hp, color 등)를 모든 클라이언트에게 전송합니다.
- 클라이언트의 위치 정보를 받아 충돌 처리 및 hp 계산, 총알들의 실시간 좌표, 게임의 상태(진행 중인지, 게임 오버인지)를 실시간으로 각 클라이언트에게 전송합니다.
- 이것을 종료할 때까지 반복합니다.

03 구조 소개

>>> High-Level Design



클라이언트

- 서버에게 접속을 요청합니다.
- ready 정보를 서버에 보내고 다른 플레이어가 ready할 때까지 기다립니다.
- 모든 플레이어가 ready한다면 게임이 시작되고, 게임 시작 초기 정보를 서버에게 수신합니다.
- 실시간으로 서버에게 나의 현재 좌표를 보내고, 서버로부터 나의 위치 및 다른 플레이어의 위치, 총알 좌표, 게임의 상태, hp를 수신해 update합니다.
- 만약 update된 게임 상태가 end면 end scene으로, 아니라면 progressing scene으로 갑니다.
- render해줍니다.
- 이것을 게임이 끝날 때까지 반복합니다.

04 주요 코드

>>> 프로토콜(서버/클라이언트)

```
typedef struct Packet
{
    int id = 0;
    SOCKET client_sock;
    Gamestate_info gameState;
    Player_Info p_info;
    Obstacle_Info o_info[MAXOBSTACLE];

}Packet;
```

서버

서버에서 클라이언트로 패킷을 전송할 때 사용하는 프로토콜입니다. SOCKET 정보와 게임 상태 정보, 플레이어의 정보, 장애물(총알)의 정보, 그리고 id값을 전송할 수 있습니다.

```
typedef struct Packet
{
    SOCKET client_sock;
    Gamestate_info gameState;
    Player_Info p_info;
    Obstacle_Info o_info[MAXOBSTACLE];
}Packet;
```

클라이언트

클라이언트에서 서버로 패킷을 전송할 때 사용하는 프로토콜입니다. SOCKET 정보와 게임 상태 정보, 플레이어의 정보, 장애물(총알)의 정보를 전송할 수 있습니다. 서버의 프로토콜과는 다르게 각각의 클라이언트에서 처리되기 때문에 id값을 받지 않습니다.

```
extern Packet g_packet;
```

04 주요 코드

>>> accept()

```
while (1) {
    // accept()
    addrlen = sizeof(clientaddr);
    g_packet[num].client_sock = accept(listen_sock, (SOCKADDR*)& clientaddr, &addrlen);
    if (g_packet[num].client_sock == INVALID_SOCKET) {
        err_display("accept()");
        break;
    }
    cout << "클라이언트 id " << num << " accept" << endl;

    // 접속한 클라 순서대로 id를 매겨준다.
    g_packet[num].id = num;

    // 순서대로 매겨준 id를 클라에 넘겨 자신의 id를 알게 한다.
    send(g_packet[num].client_sock, (char*)& num, sizeof(num), 0);

    // 접속한 클라이언트 정보 출력
    printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

    hThread[num] = CreateThread(NULL, 0, ProcessClient, (LPVOID)g_packet[num].client_sock, 0, 0);
    num++;
}
```

서버

main에서의 루프 부분입니다. accept하는 단계이며 클라이언트와 각종 정보를 받고 CreateThread를 통해 ProcessClient를 스레드로 시작시킵니다.

04 주요 코드

>>> ClientProcess()

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    SOCKET client_sock = (SOCKET)arg;
    int retval;
    int current_id = getId(client_sock);

    bool collision = false; //각각의 클라이언트가 충돌했는지 여부를 위한 지역변수

    while (true)
    {
        // 현 스레드에 연결된 클라의 게임 상태가 Win이면,
        // 나머지 클라 스레드를 재실행한다.
        // 이는 게임 종료 후 재시작 시 필요한 함수다.
        Resume_Check(current_id);


        // 현재 클라에게 준비를 받는다.
        recv_Ready(g_packet[current_id].client_sock, current_id);

        // 만약 하나라도 Progressing 상태라면 끝난게 아니므로 대기한다.
        while (true)
        {
            if (AllReady())
            {
                break;
            }// 모두 준비 되었는가?
        }

        // resume_flag가 true면(게임이 재시작 되었으면)
        // 모든 클라의 상태를 Progressing으로 바꾸고,
        // 모든 클라에게 해당 클라의 게임 상태를 보내고 flag를 false로 한다.
        // flag는 공유 변수이고 3개의 스레드 중
        // 한 개의 스레드에서만 실행되어야 하므로 임계 영역을 설정한다.
        EnterCriticalSection(&cs); // 한명의 클라이언트만 실행하도록 lock()
```

서버

각 클라이언트 정보를 따로 관리하기 위해 스레드를 사용하기로 했습니다. ProcessClient()를 스레드로 관리할 수 있습니다.

처음으로는 준비 정보를 받습니다.

04 주요 코드

>>> ClientProcess()

```
if (resume_flag) // 1,2,3 클라중에 한명만 실행하면 됨
{
    Switching_Once();
}
LeaveCriticalSection(&cs);//unlock()
cout << current_id << " 번째 게임 상태 : " << g_packet[current_id].gameState << endl;

// 현 클라의 게임 상태가 Progressing이면 계속 돈다.

//현재 클라이언트의 상태가 progressing 일때만 루프돌도록.
//실제 인게임 내의 환경임
while(g_packet[current_id].gameState == Progressing)
{
    // 플레이어의 위치 정보를 받아온다.
    recv_PlayerPosInfo(current_id);

    // 게임 재개 시, 플레이어의 포지션이 숨겨져 있었다면?

    IF_Pos_is_Hide(current_id);

    // 서버의 장애물 정보를 업데이트한다.
    //공유자원이므로 임계영역으로 처리함.
    EnterCriticalSection(&cs);
    obstacleUpdate();
    LeaveCriticalSection(&cs);

    // 충돌 체크와 Hp 업데이트를 진행한다.
    CollisionCheck_And_HpUpdate(current_id, collision);

    // Hp 업데이트된 부분이 있을지도 모르니 게임 상태를 업데이트한다.
    gameStateUpdate(current_id);

    // 이긴 사람이 있는지 확인하여 Win상태로 만들 수 있으면 그렇게 한다.
    checkWin();
```

서버

이전 페이지의 코드 마지막 줄에서 CriticalSection으로 들어가는 코드가 있습니다. 재개를 위한 resume_flag가 공유 자원이기 때문에 임계영역으로 lock, unlock을 통해 사용해야 합니다.

Progressing 루프일 때부터 게임 시작입니다. 플레이어의 정보와 장애물 정보를 받고, 충돌 체크 및 hp 업데이트를 해준 다음, 게임 상태 업데이트 및 이긴 사람이 있는지 체크합니다.

obstacleUpdate()에서 obstacle도 공유 자원이기 때문에 임계영역을 설정해 사용하는 것을 보실 수 있습니다.

04 주요 코드

>>> ClientProcess()

```
            // 모든 정보를 보내기 전에, 패킷에 장애물 정보를 Set해준다.
            SetBeforeSendObstacleInfo();

            // 모든 정보를 보낸다.
            send_Allinfo(current_id, collision);
        }

        // 현 클라가 Win이 아니면 기다려야 하니 스레드를 일시 정지시킨다.
        WaitForRemainder(current_id);

        // Win인 클라는 WaitForRemainder함수를 거치지 않고
        // 모든 정보를 reset한 후 루프 위로 올라간다.
        resetInfo(current_id);
    }
    return 0;
}
```

서버

업데이트된 정보들을 g_packet에 set해준 뒤, 모든 정보를 보내도록 합니다.

게임이 끝나면 스레드를 일시 정지시키고, 게임의 모든 정보를 초기화합니다.

04 주요 코드

>>> recvn()

```
int recvn(SOCKET s, char* buf, int len, int flags)
{
    int received;
    char* ptr = buf;
    int left = len;

    while (left > 0) {
        received = recv(s, ptr, left, flags);
        if (received == SOCKET_ERROR)
            return SOCKET_ERROR;
        else if (received == 0)
            break;
        left -= received;
        ptr += received;
    }

    return (len - left);
}
```

서버/클라이언트

서버와 클라이언트 둘 다 recvn()를 이용해 패킷을 수신합니다.
클라이언트 소켓 정보와 어떤 버퍼에 저장할 것인지 정하고, left가 0 초과일 때만 recv() 및 left와 received를 활용해 패킷의 길이만큼 나눠서 받고 처리할 수 있도록 합니다.

04 주요 코드

>>> Ready(Send/Recv)

```
void recv_Ready(SOCKET client_sock, int id)
{
    int retval;
    retval = recvn(client_sock, (char*)&g_saveReady[id], sizeof(int), 0);
    cout << id << " 번째 클라이언트 recv_ready" << endl;
}
```

서버

서버에서 클라이언트의 ready 정보를 받기 위해 사용할 함수입니다. recvn 함수를 이용해 해당 클라이언트 소켓으로부터 정보를 받아 g_saveReady 배열에 저장합니다.

```
void send_Ready()
{
    send(g_packet.client_sock, (char*)& g_packet.gameState, sizeof(Gamestate_info), 0);
}
```

클라이언트

클라이언트에서 서버로 ready 정보를 전송하기 위해 사용할 함수입니다. 클라이언트 소켓 정보를 설정하고 현재 g_packet에 저장되어 있는 게임 상태를 전송합니다.

04 주요 코드

>>> PosInfo(Send/Recv)

```
void recv_PlayerPosInfo(int id)
{
    // 각 플레이어의 포지션을 받아 온다.
    int retval;
    retval = recvn(g_packet[id].client_sock, (char*)&g_packet[id].p_info.position, sizeof(POSITION), 0);
}
```

서버

서버에서 클라이언트의 위치 정보를 받기 위해 사용할 함수입니다. recvn 함수를 이용해 해당 클라이언트 소켓으로부터 정보를 받아 g_packet배열의 p_info.position에 저장합니다.

```
void send_PosInfo()
{
    send(g_packet.client_sock, (char*)& player[p_num]->pos, sizeof(POSITION), 0);
}
```

클라이언트

클라이언트에서 서버로 위치 정보를 전송하기 위해 사용할 함수입니다. 클라이언트 소켓 정보를 설정하고 현재 플레이어를 관리하는 배열에서 자신의 정보에 저장되어 있는 위치를 전송합니다.

04 주요 코드

>>> hpUpdate() / obstacleUpdate()

```
Obstacle_Info hpUpdate(Obstacle_Info& o_info, int id)
{
    // id번째 플레이어의 hp를 i번째 장애물의 damage만큼 깎아준다.
    g_packet[id].p_info.hp -= o_info.damage;

    // 한 번 hp를 깎은 장애물은 초기화될 때까지 damage를 0으로 한다.
    o_info.damage = 0;

    return o_info;
}
```

```
void obstacleUpdate()
{
    // 각 장애물들을 업데이트 처리해준다.
    for (int i = 0; i < MAXOBSTACLE; ++i)
    {
        g_obstacleManager[i].position.y -= g_obstacleManager[i].speed;

        //장애물의 스폰을 위함임
        if (g_obstacleManager[i].position.y < -100)
        {
            g_obstacleManager[i].position.y = (float)uidY(dre);
            g_obstacleManager[i].damage = 10;
        }
    }
}
```

```
void CollisionCheck_And_HpUpdate(int id, bool& collision)
{
    Player_Info p_info;
    p_info = g_packet[id].p_info;
    Obstacle_Info o_info;

    // 업데이트된 g_obstacleManager를 하나씩 가져와 o_info에 넣고,
    // 위의 p_info와 함께 Collision_Check함수의 인자에 넣어 충돌 체크한다.
    collision = false;
    for (int i = 0; i < MAXOBSTACLE; ++i)
    {
        o_info = g_obstacleManager[i];

        if (Collision_Check(p_info.position.x, p_info.position.y,
            o_info.position.x, o_info.position.y,
            p_info.size, o_info.size))
        {
            // 충돌 체크되었다면 hp를 업데이트한다.
            g_obstacleManager[i] = hpUpdate(o_info, id);
            collision = true;

        }
    }
}
```

서버

hpUpdate()에서는 장애물 정보와 클라이언트 id를 받아와 해당 클라이언트의 hp를 업데이트합니다.

obstacleUpdate()에서는 장애물의 위치 및 리스폰을 업데이트합니다.

collisionCheck와 hpUpdate를 같이 진행합니다.

obstacleUpdate()는 ClientProcess 스레드에서 진행합니다.

05 느낀 점

>>> 개발 당시 문제점과 해결 방안

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    SOCKET client_sock = (SOCKET)arg;
    int retval;
    int current_id = getId(client_sock);

    bool collision = false; //각각의 클라이언트가 충돌했는지 여부를 위한 지역변수

    while (true)
    {
        // 현 스레드에 연결된 클라의 게임 상태가 Win이면,
        // 나머지 클라 스레드를 재실행한다.
        // 이는 게임 종료 후 재시작 시 필요한 함수다.
        Resume_Check(current_id);


        // 현재 클라에게 준비를 받는다.
        recv_Ready(g_packet[current_id].client_sock, current_id);

        // 만약 하나라도 Progressing 상태라면 끝난게 아니므로 대기한다.
        while (true)
        {
            if (AllReady())
            {
                break;
            }// 모두 준비 되었는가?
        }

        // resume_flag가 true면(게임이 재시작 되었으면)
        // 모든 클라의 상태를 Progressing으로 바꾸고,
        // 모든 클라에게 해당 클라의 게임 상태를 보내고 flag를 false로 한다.
        // flag는 공유 변수이고 3개의 스레드 중
        // 한 개의 스레드에서만 실행되어야 하므로 임계 영역을 설정한다.
        EnterCriticalSection(&cs); // 한명의 클라이언트만 실행하도록 lock()
```

앞서 설명했던 ProcessClient()의 구조를 만들 때 팀원 3명이 평일에 매일 공강 시간에 모여 머리를 맞대며 회의했던 기억이 남아 있습니다. 서버와 클라이언트의 데이터 송신/수신 순서를 계속 체크하면서 작업해야 했기에 서로의 작업 진행 상황을 공유하고, 수정할 곳이나 보완할 곳이 생겨나면 수정/보완 작업을 거쳤습니다.

요점은 주석입니다. 첫 개발 시점에는 팀원들끼리 주석을 달지 않고 내용을 공유해 단시간 코드 파악에 어려움이 있었습니다. 그 뒤로 협업할 때 어떤 함수가 어떤 순서로 동작하며 어떤 역할을 하는지에 대해 자세하게 주석 처리하자는 의견이 나왔고, 실제로 그렇게 개발을 진행하니 능률이 올라갔단 느낌을 팀원 전체가 받았습니다. 당시 조금 과할 정도로 주석을 자세하게 달아 놓기는 했지만, 한 번 주석을 달아 놓으면 확실히 어느 라인까지 도달했을 때 어느 데이터가 오고 갔는지를 단번에 확인할 수 있어 전에 개발했을 때보다 훨씬 수월했습니다.