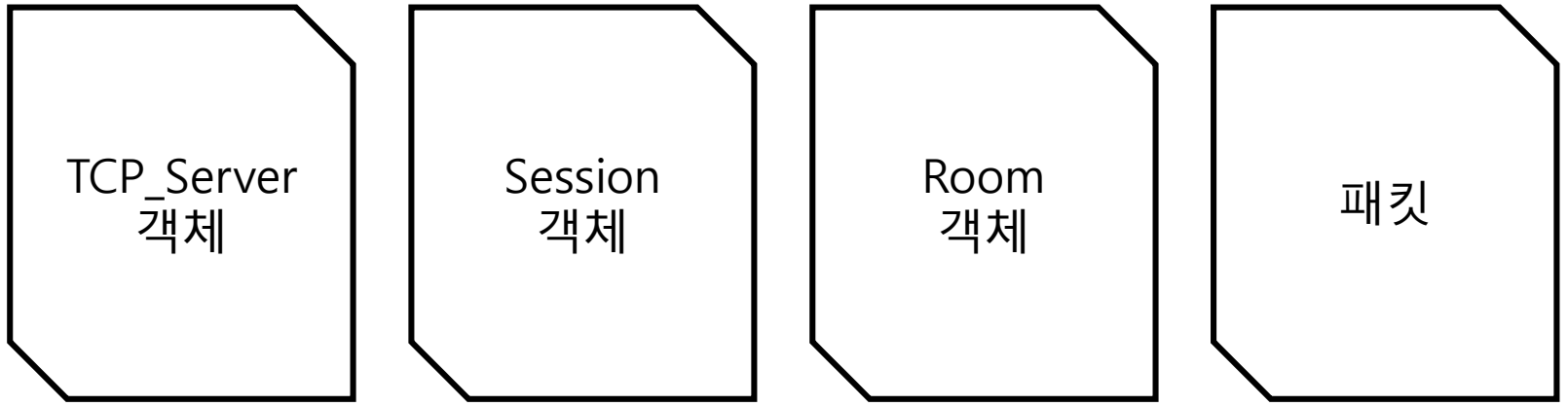
언리얼 엔진 기반
체스 게임 구현

【 목차 】

【 서버 구성 】



【 서버 구성 - TCP_Server 】

```cpp
class TCP_Server : public std::enable_shared_from_this<TCP_Server>
{
public:
    TCP_Server(boost::asio::io_service& io_service) :
        m_io_service(io_service),
        m_Acceptor(io_service, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), PORT_NUMBER))
    {
        CreateRoom("Lobby");
    }

    ~TCP_Server()
    {
        Stop();
    }

public:
    void Start();
    void Stop();
    void CloseSession(std::shared_ptr<class Session> pSession);

private:
    void StartAccept();
    void HandleAccept(std::shared_ptr<class Session> pSession, const boost::system::error_code& error);

private:
    boost::asio::io_service& m_io_service;
    boost::asio::ip::tcp::acceptor m_Acceptor;

    boost::thread_group ThreadGroup;
    const int ThreadNum = 5;

public:
    int CreateRoom(const std::string& room_name);
    std::shared_ptr<class Room> GetRoom(int room_idx);
    void GetAllRoom(std::vector<PACKET_DATA::SessionRoom>& RoomList);
    void DeleteRooms();

public:
    bool IsDuplicatedID(const std::string id);

private:
    std::unordered_set<std::shared_ptr<class Session>> m_vSessions;
    std::map<int, std::shared_ptr<class Room>> m_RoomMap;
};
```

- Session과 Room을 생성 및 관리
- boost::asio::io_service을 통해 Socket과 바인드하여 socket의 이벤트를 처리한다.
- boost::asio::ip::tcp::acceptor의 async_accept()를 통해 소켓 연결 시도
  - 성공 시, 생성한 Session을 unordered_set에 저장하여 관리
  - 생성된 Session은 패킷 수신 대기
- boost::thread_group을 통해 멀티쓰레딩 준비

【 서버 구성 - Session 】

```
class Session : public std::enable_shared_from_this<Session>
{
public:
    Session(boost::asio::io_context& io_context, std::shared_ptr<class TCP_Server> Owner) :
        m_Socket(io_context),
        m_Strand(boost::asio::make_strand(io_context)),
        m_cServer(Owner)
    {
        m_bDisconnected.exchange(false);
        memset(m_cBuffer, '\0', sizeof(m_cBuffer));
    }

    ~Session()
    {
        std::cout << "{" << this << "} Session is Destoryed" << std::endl;
    }

public:
    void ASyncRead();
    void ASyncWrite(class Buffer& buf);
    void Disconnect();

private:
    void HandleRead(const boost::system::error_code& error, size_t bytes_transferred);
    void HandleWrite(const boost::system::error_code& error, size_t bytes_transferred);

public:
    boost::asio::ip::tcp::socket& GetSocket() { return m_Socket; }

private:
    std::atomic<bool> m_bDisconnected;
    boost::asio::ip::tcp::socket m_Socket;
    boost::asio::strand<boost::asio::io_context::executor_type> m_Strand;

    std::weak_ptr<class TCP_Server> m_cServer;
    char m_cBuffer[PACKET_SIZE];

private:
    std::weak_ptr<class Room> m_cRoom;
};
```

- 생성 시 TCP_Server의 boost::asio::io_service을 통해 boost::asio::ip::tcp::socket 및 boost::asio::strand 생성

- boost::asio::ip::tcp::socket을 통해 연결된 클라이언트와 TCP 통신
  - async_read_some()을 통해 비동기식 수신 작업을 수행
    - 수신 실패 시, 연결 종료
    - 수신 성공 시, 수신한 패킷 분석 후 작업을 수행
    - 패킷 작업을 마친 후, 비동기식 수신 대기

  - async_write_some()을 통해 비동기식 송신 작업을 수행
    - 송신 실패 시, 연결 종료
    - 송신 성공 시, 비동기식 수신 대기

【 서버 구성 - Session 】

- 패킷 수신 시, 패킷 분석 후 작업 수행
  - o 수신한 패킷의 아이디를 기반으로 작업 수행
  - o Ex) PACKET_ID == PT_ENTER_ROOM → ResponseEnterRoom() 함수 수행
    - 해당하는 Room에 Session을 추가 후 요청한 클라이언트에 ACK 및 PT_RES_PLAYER_LIST 패킷 전송
    - Lobby에 위치한 클라이언트들에 PT_RES_ROOM_LIST 패킷 전송

- boost::asio::strand<boost::asio::io_context::executor_type>을 통해
  멀티쓰레드 프로그램에서 명시적인 잠금 없이 쓰레드를 사용할 수 있도록 함
  - o boost::asio::bind_executor()를 통해 Strand에 함수를
    바인드하여 쓰레드 안전 접근을 보장한다.

- 하나의 Session이 하나의 클라이언트와 통신하기 때문에 각각의 Session 마다 Buffer를 멤버 변수로 소유

【 서버 구성 - Room 】

```
class Room
{
public:
    Room(const int room_id, const std::string name) :
        m_strRoomName(name),
        m_iRoomID(room_id) { }

public:
    void RequestGameStart(std::shared_ptr<class Session> pSession, uint8_t state);
    void GameStart();

    void RequestNextRound(std::shared_ptr<class Session> pSession, uint8_t curr_round);
    void NextRound();

    void RequestGameEnd(std::shared_ptr<class Session> pSession);
    void GameEnd();

public:
    void Enter(std::shared_ptr<class Session> pSession);
    void Leave(std::shared_ptr<class Session> pSession);
    void Broadcast(class Buffer& buf);

public:
    bool IsPossibleGameStart() const;
    bool IsPossibleChangeRound(const uint8_t round) const;
    bool IsPossibleGameEnd() const;
    bool IsPlayingGame() const { return (m_iRoomID > 0 && game_round > 0); }

private:
    int m_iRoomID;
    std::string m_strRoomName;

private:
    struct SESSION_STATE
    {
        uint8_t state;
        uint8_t round = 0;
    };

    std::map<std::shared_ptr<class Session>, SESSION_STATE> m_PlayerMap;
    uint8_t game_round = 0;
};
```

- Room은 id, 이름, 구성 인원으로 구성됨
  - 각 구성 인원은 Ready 상태 여부와 게임 라운드 정보를 저장
- 최대 인원 수는 2명이며, 2명이 모두 Ready 상태일 경우 게임 시작
- Broadcast() 함수를 구성하여 Room 내의 모든 인원에게 패킷을 송신할 수 있도록 함
- 게임 진행 중일 경우, RequestGameStart(), RequestNextRound(), RequestGameEnd() 를 통해 Room의 상태를 업데이트하도록 함
  - Room의 모든 인원이 상태 변경을 요청할 경우, GameStart(), NextRound(), GameEnd()의 상태 업데이트 진행

【 서버 구성 - Packet - 패킷 구성 】

```
struct PACKET_HEADER
{
    uint16_t content_length;
    uint32_t packet_id;
    uint8_t sequence_num;
};
```

```
enum PACKET_ID : uint32_t
{
    PT_PROGRAM_START,
    PT_CHAT,
    PT_REQ_ROOM_LIST,
    PT_RES_ROOM_LIST,
    PT_REQ_PLAYER_LIST,
    PT_RES_PLAYER_LIST,
    PT_CREATE_ROOM,
    PT_ENTER_ROOM,
    PT_LEAVE_ROOM,
    PT_UPDATE_PLAYER_STATE,
    PT_GAME_START,
    PT_REQ_NEXT_ROUND,
    PT_RES_NEXT_ROUND,
    PT_GAME_END,
    PT_CHESS_MOVE,
    PT_CHESS_CASTLING,
    PT_CHESS_PROMOTION
};
```

- **패킷의 크기**는 1024 BYTE 이다.

- **PacketHeader**
  - content_length를 통해 송수신한 패킷에 이상이 없는 지 확인한다.
  - packet_id를 통해 송수신한 패킷의 작업을 분별한다.
  - sequence_num은 송수신하는 패킷의 길이가 짧아 거의 사용되지 않는다.
    - Room_List와 Player_List를 송신하는 경우,
      많은 양의 데이터를 전송 시, List를 분리하여
      다수 개의 패킷을 보낸다.
      이때, 다수 개의 패킷을 분별하기 위해 사용한다.

【 서버 구성 - Packet - 패킷 구성 】

다음의 PACKET_ID에 따라 아래와 같은 패킷 데이터를 요청한다.

```
enum PACKET_ID : uint32_t
{
    PT_PROGRAM_START,
    PT_CHAT,
    PT_REQ_ROOM_LIST,
    PT_RES_ROOM_LIST,
    PT_REQ_PLAYER_LIST,
    PT_RES_PLAYER_LIST,
    PT_CREATE_ROOM,
    PT_ENTER_ROOM,
    PT_LEAVE_ROOM,
    PT_UPDATE_PLAYER_STATE,
    PT_GAME_START,
    PT_REQ_NEXT_ROUND,
    PT_RES_NEXT_ROUND,
    PT_GAME_END,
    PT_CHESS_MOVE,
    PT_CHESS_CASTLING,
    PT_CHESS_PROMOTION
};
```

[패킷 아이디]

```
struct ROOM_INFO
{
    PACKET_DATA::SessionRoom room;
};
struct ROOM_LIST_INFO
{
    uint8_t room_cnt;
    std::vector<SessionRoom> RoomList;
};
struct PLAYER_INFO
{
    PACKET_DATA::SessionPlayer player;
};
struct PLAYER_LIST_INFO
{
    uint8_t player_cnt;
    std::vector<PACKET_DATA::SessionPlayer> PlayerList;
};
struct GAME_INFO
{
    uint8_t round;
    PACKET_DATA::SessionPlayer WhiteTeamPlayer;
    PACKET_DATA::SessionPlayer BlackTeamPlayer;
};
```

[Room/Player/Game 데이터]

```
struct CHESS_MOVE
{
    uint8_t round;
    uint8_t src_slot_index;
    uint8_t dest_slot_index;
};
struct CHESS_CASTLING
{
    uint8_t round;
    uint8_t king_slot_index;
    uint8_t rook_slot_index;
};
struct CHESS_PROMOTION
{
    uint8_t round;
    uint8_t pawn_slot_index;
    uint8_t promotion_type;
};
```

[체스 이벤트 데이터]

```
struct SessionRoom
{
    uint8_t index;
    char name[ROOM_NAME_SIZE];
    uint8_t participant_num;
};
```

[Room Info]

```
struct SessionPlayer
{
    enum STATE_IN_ROOM : uint8_t
    {
        NONE,
        NOT_READY,
        READY,
    };

    char id[SESSION_ID_SIZE];
    uint8_t state;
};
```

[Player Info]

【 서버 구성 - Packet – 패킷 핸들러 】

```cpp
void NSW_PacketHandler::PackRoomInfoPacket(Buffer& SendBuffer, const uint32_t pkt_id, const uint8_t room_index, const std::string room_name, const uint8_t participants)
{
    PACKET_HEADER pkt_header;
    PACKET_DATA::ROOM_INFO pkt_data;

    pkt_data.room.index = room_index;
    strcpy(pkt_data.room.name, room_name.c_str());
    pkt_data.room.participant_num = participants;

    pkt_header.content_length = sizeof(pkt_data);
    pkt_header.packet_id = pkt_id;
    pkt_header.sequence_num = 0;

    SendBuffer << pkt_header;
    SendBuffer << pkt_data;
}
```

- **패킷 핸들러**를 통해 각 패킷 데이터를 기반으로 버퍼에 패킷 데이터를 직렬화
  - 각 PACKET_ID에 해당하는 데이터를 매개변수로 입력 받아 패킷 데이터를 구성한다.
  - 구성된 패킷 데이터를 참고하여 패킷 헤더를 구성한다.
  - 버퍼에 구성한 패킷 헤더, 패킷 데이터 순으로 직렬화한다.

【 서버 구성 - Packet – 패킷 핸들러 】

```cpp
void NSW_PacketHandler::UnpackHeader(Buffer& RecvPacket, uint16_t& content_length, uint32_t& pkt_id, uint8_t& seq_num)
{
    PACKET_HEADER pkt_header;
    RecvPacket >> pkt_header;

    content_length = pkt_header.content_length;
    pkt_id = pkt_header.packet_id;
    seq_num = pkt_header.sequence_num;
}

uint16_t NSW_PacketHandler::UnpackRoomInfoPacket(Buffer& RecvPacket, uint8_t& room_index, std::string& room_name, uint8_t& participants)
{
    PACKET_DATA::ROOM_INFO pkt_data;
    RecvPacket >> pkt_data;

    room_index = pkt_data.room.index;
    room_name = pkt_data.room.name;
    participants = pkt_data.room.participant_num;

    return sizeof(pkt_data);
}
```

- **패킷 핸들러**를 통해 각 패킷 데이터를 기반으로 버퍼의 데이터를 역직렬화
  - 버퍼에 저장된 데이터를 직렬화한 순서와 동일하게
    패킷 헤더, 패킷 데이터 순으로 역직렬화한다.
  - 패킷 데이터의 값들은 참조 매개변수를 통해 전달한다.
  - 수신한 패킷이 유효한 지 확인하기 위한 Content_length를 리턴한다.

【 서버 구성 - Packet - Buffer 객체 】

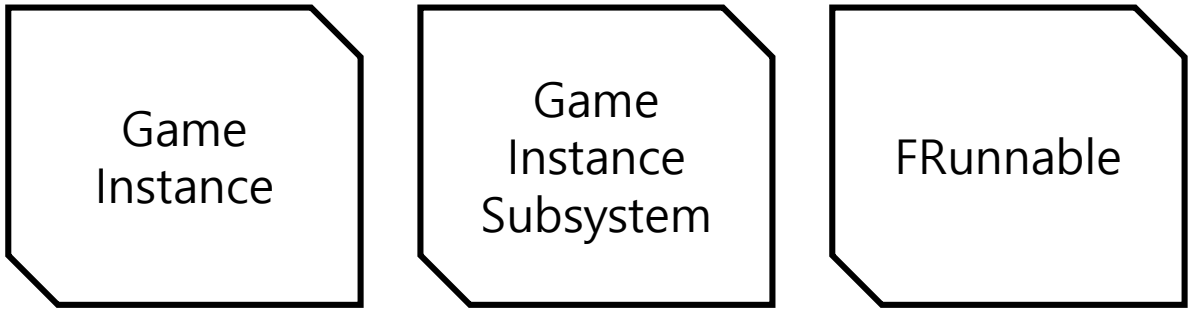
```cpp
// Serialize
template<typename T>
Buffer& operator << (const T& in)
{
    if (m_nOffset >= 0)
    {
        memcpy(m_Buffer + m_nOffset, &in, sizeof(in));
        m_nOffset += sizeof(in);
    }

    return *this;
}

// Deserialize
template<typename T>
Buffer& operator >> (T& out)
{
    if (m_nOffset >= 0)
    {
        memcpy(&out, m_Buffer + m_nOffset, sizeof(out));
        m_nOffset += sizeof(out);
    }

    return *this;
}
```

- Session에서 송수신한 버퍼의
  직렬화 및 역직렬화하기 위한 객체이다.

- Buffer는 총 1024 BYTE 크기의 char 배열을 지닌다.

- 연산자 << 와 >> 를 통해
  Buffer에 값을 추가하거나 값을 추출한다.

【 클라이언트 구성 】



【 클라이언트 구성 - GameInstance 】

```
class CHESSPROJECT_API UChessGameInstance : public UGameInstance
{
public:
    virtual void Init() override;
    virtual void Shutdown() override;

private:
    bool Tick(float DeltaSeconds);

public:
    void ProgramStart(const FText PlayerID);
    void ShowChat(const FString ChatMessage);
    void PreUpdateLobby();
    void UpdateLobby(const std::vector<PACKET_DATA::SessionRoom>& RoomList);
    void UpdateRoom(const std::vector<PACKET_DATA::SessionPlayer>& PlayerList);
    void EnterRoom(const FText RoomName);
    void EnterLobby();
    void UpdatePlayerState(const FText PlayerID, const uint8_t PlayerState);
    void GameStart(const uint8_t GameRound, const FText WhiteTeamPlayer, const FText BlackTeamPlayer);
    void UpdateRound(int round);
    void GameEnd(const std::string room_name);
    void ProcessChessPieceMove(const uint8_t round, const uint8_t src_slot_index, const uint8_t dest_slot_index);
    void ProcessCastling(const uint8_t round, const uint8_t king_slot_index, const uint8_t rook_slot_index);
    void ProcessPromotion(const uint8_t round, const uint8_t pawn_slot_index, const uint8_t promotion_type);
    void ProgramEnd();

public:
    void HideAllWidget();
    void ShowProgramStartWidget();
    void ShowLobbyWidget();
    void ShowPromotionWidget();
    void ShowRoomWidget();
    void ShowInGameWidget();
    void ShowGameResultWidget(const FText game_state, const FText game_result);

private:
    FTSTicker::FDelegateHandle TickDelegateHandle;
    GAME_STATE m_SessionState;
    FText m_PlayerID;
    bool bChangeTurn;

private:
    class UProgramStartWidget* m_ProgramStartWidget;
    class ULobbyWidget* m_LobbyWidget;
    class URoomWidget* m_RoomWidget;
    class UInGameWidget* m_InGameWidget;
    class UUserWidget* GameResultWidget;
};
```

```
enum GAME_STATE : uint8_t
{
    INIT,
    IN_LOBBY,
    IN_ROOM,
    IN_GAME,
    END
};
```

- 서버에서 수신 받은 패킷을 실질적으로 처리
  - 프로그램 시작, 로비, 방 및 플레이어 정보 업데이트, 게임 시작 및 종료 등

- FTSTicker::FDelefateHandle에 해당 클래스 내에서 정의한 Tick() 함수를 바인딩해 **Tick 이벤트** 처리
  - **Tick 이벤트**: 서버와의 연결 확인 후, 수신 받은 패킷이 존재하면 해당 패킷 이벤트 처리

- 플레이어 정보 관리 (게임 상태, 이름 등)

- 위젯 관리 (시작, 로비, 방, 게임 등의 UI)

【 클라이언트 구성 - GameInstanceSubSystem 】

```
class CHESSPROJECT_API UMyNetworkSubsystem : public UGameInstanceSubsystem
{
public:
    virtual void Initialize(FSubsystemCollectionBase& Collection) override;
    virtual void Deinitialize() override;

public:
    void ConnectToServer(FString Address, int port);

    void DisconnectToServer();

public:
    void RequestProgramStart(const FString PlayerID);
    void RequestCreateRoom(const FString RoomName);
    void RequestEnterRoom(const uint8_t RoomIndex);
    void RequestUpdatePlayerState(const uint8_t player_state);
    void RequestUpdateRoom();
    void RequestUpdateLobby();
    void RequestNextRound(const uint8_t round);
    void RequestChessMove(const uint8_t src_slot_index, const uint8_t dest_slot_index);
    void RequestChessCastling(const uint8_t king_slot_index, const uint8_t rook_slot_index);
    void RequestChessPromotion(const uint8_t pawn_slot_index, const uint8_t promotion_type);
    void RequestGameEnd();

public:
    void ResponsePacket();

private:
    void ResponseProgramStart(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseChat(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseUpdateLobby(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseUpdateRoom(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseCreateRoom(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseEnterRoom(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseEnterLobby(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseUpdatePlayerState(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseGameStart(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseNextRound(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseGameEnd(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseMovingChessPiece(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponseCastlingChessPiece(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);
    void ResponsePromoteChessPiece(class Buffer& RecvPacket, const uint16_t& content_length, const uint8_t& seq_num);

private:
    class FSocket* m_Socket;

    TSharedPtr<class FRecvWorker> RecvWorkerThread;
    TSharedPtr<class FSendWorker> SendWorkerThread;
};
```

- **서브시스템**은 수명이 관리되는 자동 인스턴싱 클래스
  - 엔진에서 인스턴싱하고 생성 및 소멸을 관리

- 서버와의 소켓 통신을 수행하기 위한 클래스
  - FRunnable을 상속한 FRecvWorker & FSendWorker 클래스를 통해 서버와의 소켓 통신을 수행
  - **Request 함수**들을 통해 서버에 데이터 요청
  - **Response 함수**들을 통해 수신한 패킷 처리
    - ResponsePacket() 함수를 통해 수신한 패킷의 헤더를 확인 후 PACKET_ID에 해당하는 Response 함수들을 실행한다.

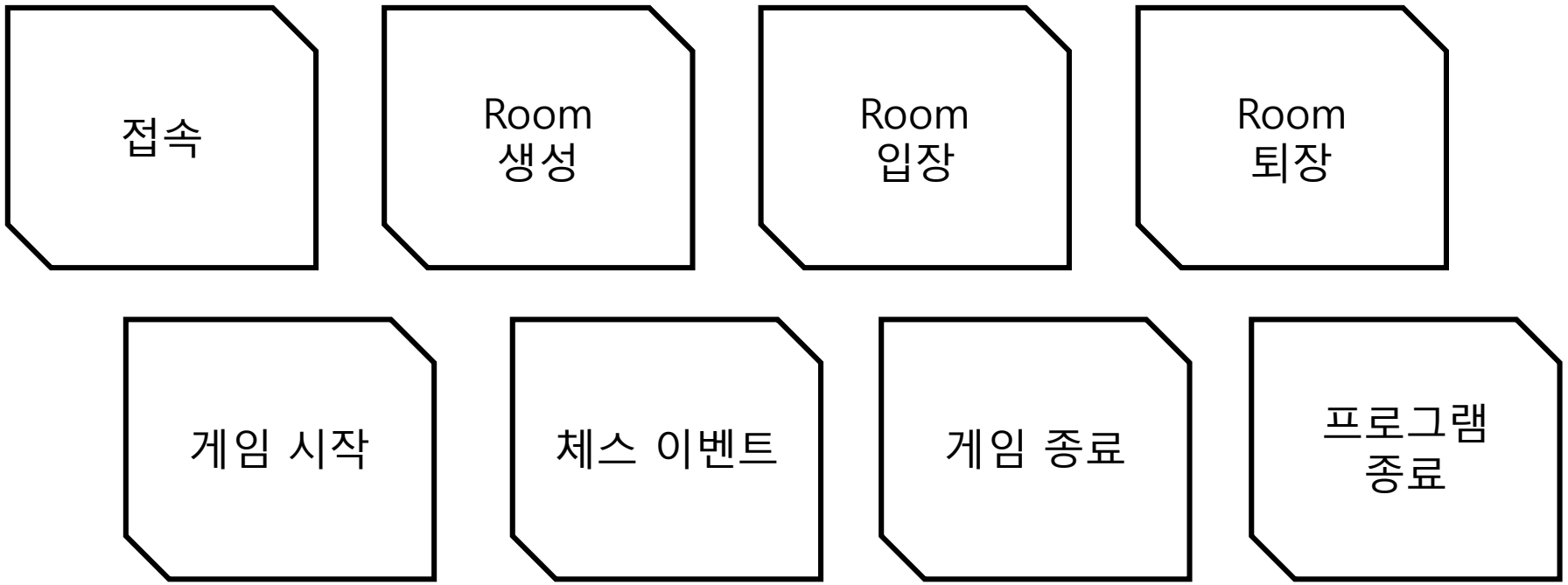
【 클라이언트 구성 - RecvWorker & SendWorker 】

```cpp
class CHESSPROJECT_API FRecvWorker : public FRunnable
{
public:
#pragma region Main Thread Code
    FRecvWorker(class FSocket* Socket) : m_Socket(Socket)
    {
        m_RecvThread = FRunnableThread::Create(this, TEXT("RecvThread"));
    }

    ~FRecvWorker()
    {
        if (m_RecvThread)
        {
            m_RecvThread->WaitForCompletion();
            m_RecvThread->Kill();
            delete m_RecvThread;
            m_RecvThread = nullptr;
        }
    }
#pragma endregion

public:
    virtual bool Init() override;
    virtual uint32 Run() override;
    virtual void Exit() override;
    virtual void Stop() override;

public:
    bool ReadPacketQueue(class Buffer& buf);

private:
    bool RecvPacket(uint8* Results, int32 Size);

private:
    FRunnableThread* m_RecvThread;
    class FSocket* m_Socket;
    TQueue<TArray<uint8>> RecvPacketQueue;
    bool m_bRunning;
};
```

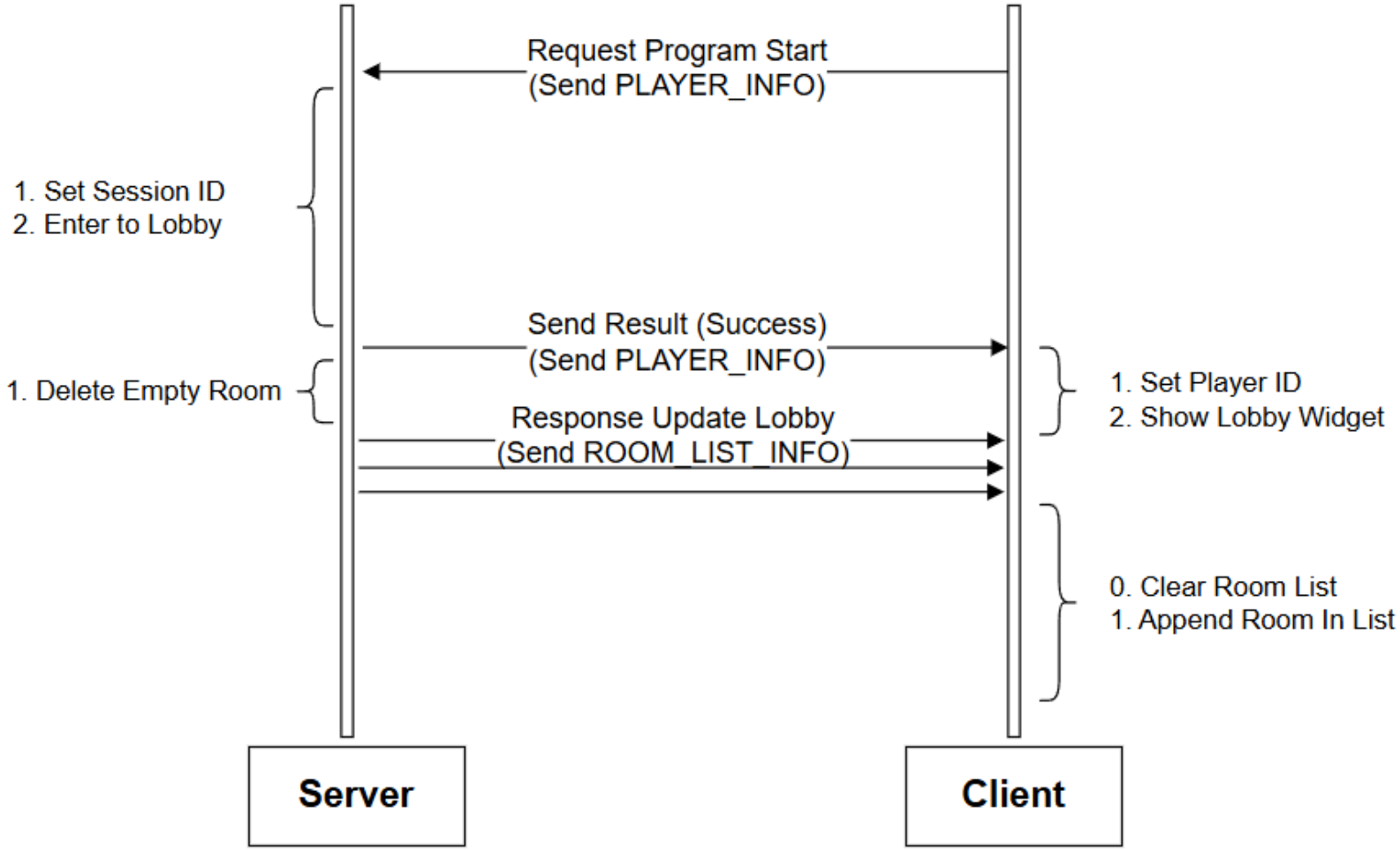
```cpp
class CHESSPROJECT_API FSendWorker : public FRunnable
{
public:
#pragma region Main Thread Code
    FSendWorker(class FSocket* Socket): m_Socket(Socket)
    {
        m_SendThread = FRunnableThread::Create(this, TEXT("SendThread"));
    }

    ~FSendWorker()
    {
        if (m_SendThread)
        {
            m_SendThread->WaitForCompletion();
            m_SendThread->Kill();
            delete m_SendThread;
            m_SendThread = nullptr;
        }
    }
#pragma endregion

public:
    virtual bool Init() override;
    virtual uint32 Run() override;
    virtual void Exit() override;
    virtual void Stop() override;

public:
    bool WritePacketQueue(class Buffer& buf);

private:
    bool SendPacket(TArray<uint8>& buf);

private:
    FRunnableThread* m_SendThread;
    class FSocket* m_Socket;
    TQueue<TArray<uint8>> SendPacketQueue;
    bool m_bRunning;
};
```

- FRunnable : 임의의 쓰레드에서 실행되는 클래스
  - Init()
    - § 게임 쓰레드에서 실행
    - § 멤버 변수 초기화
    - § 리턴 값을 통해 쓰레드 실행 결정
  - Run()
    - ▪ 새로 생성한 쓰레드에서 실행
    - ▪ 멀티쓰레딩 작업 수행
    - ▪ RecvPacket() & SendPacket() 수행
  - Exit()
    - ▪ 새로 생성한 쓰레드에서 실행.
    - ▪ Run() 리턴 시 호출되어 Clean-Up
  - Stop()
    - ▪ 게임 쓰레드에서 실행
    - ▪ 원하는 타이밍에 호출하여 Run() 종료

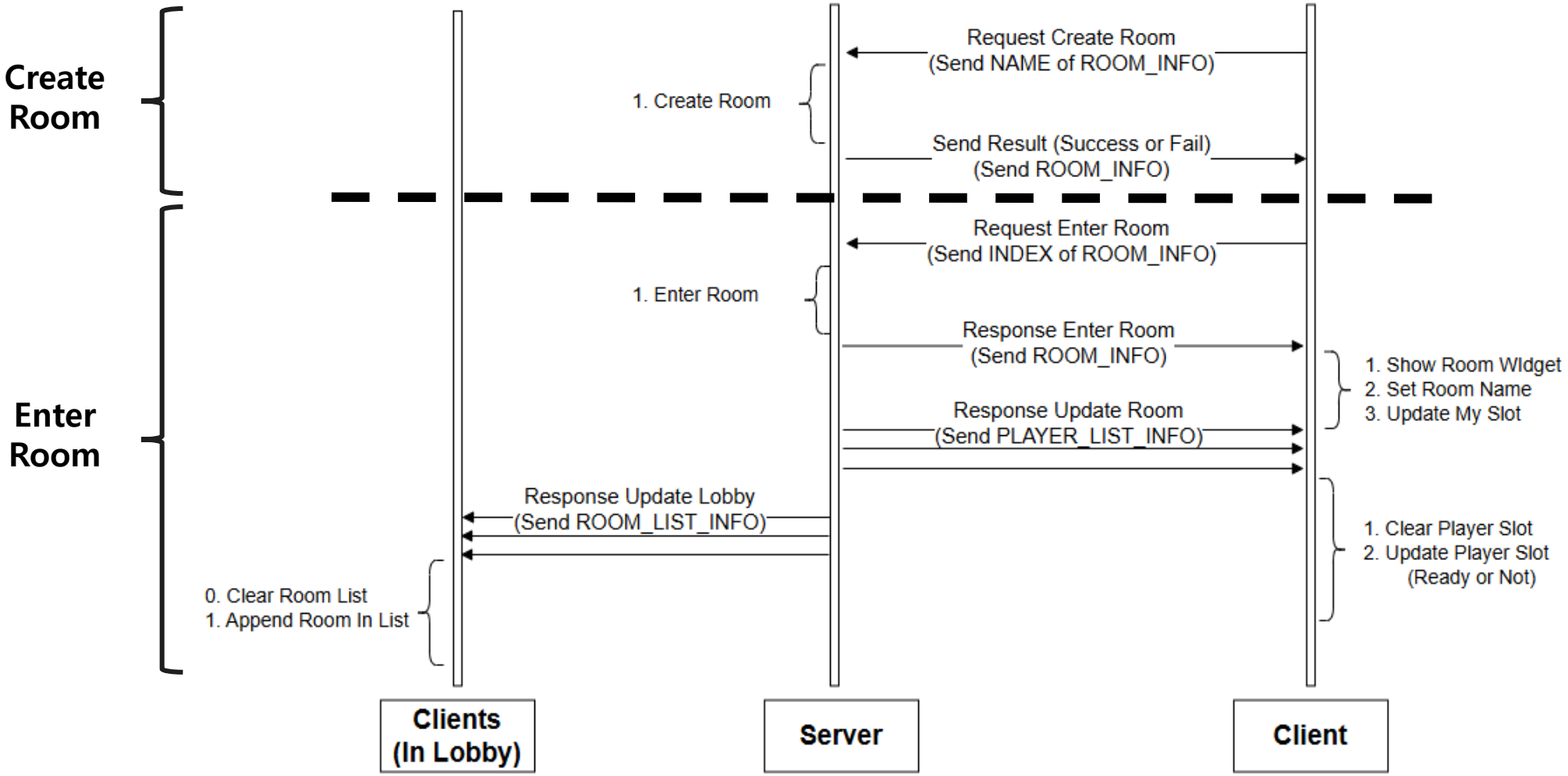
【 이벤트 다이어그램 】



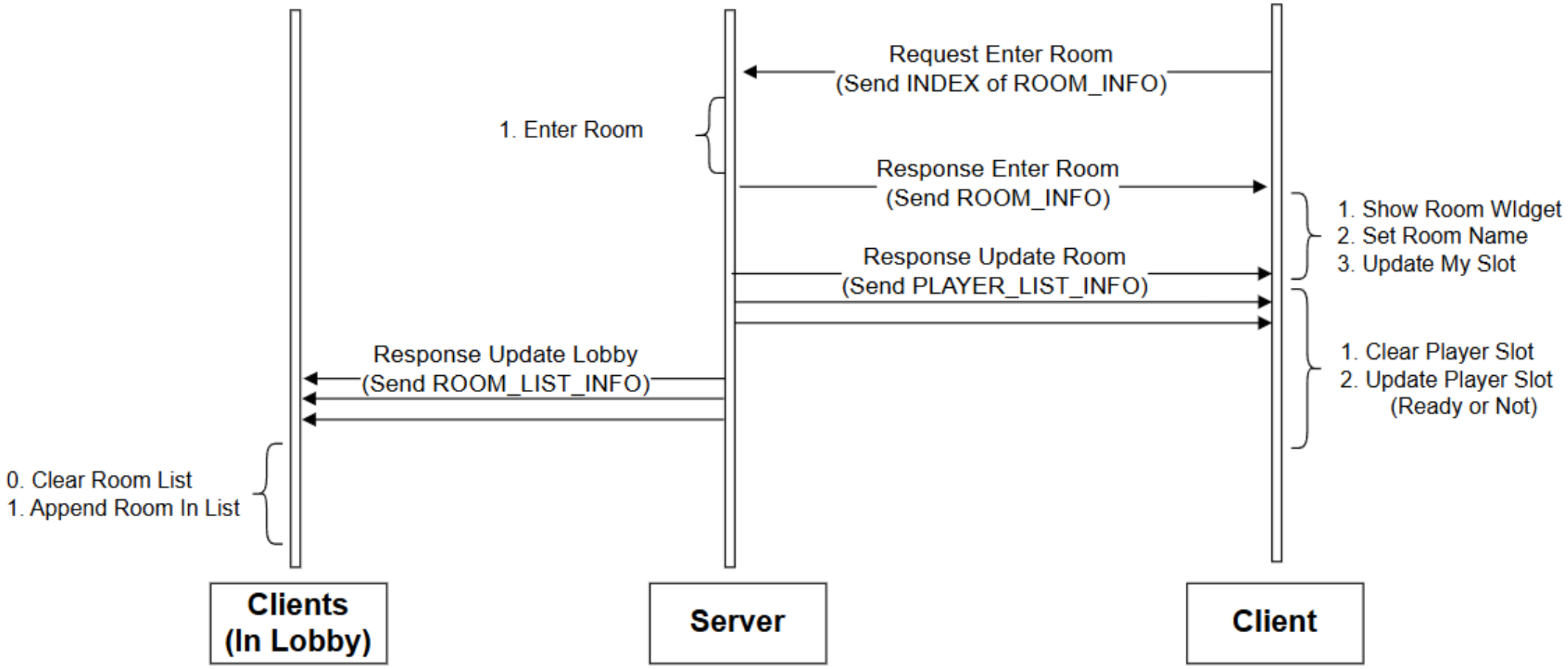
【 이벤트 다이어그램 - 접속 】


Request Program Start
(Send PLAYER_INFO)
1. Set Session ID
2. Enter to Lobby
Send Result (Success)
(Send PLAYER_INFO)
1. Delete Empty Room
1. Set Player ID
2. Show Lobby Widget
Response Update Lobby
(Send ROOM_LIST_INFO)
0. Clear Room List
1. Append Room In List
Server
Client

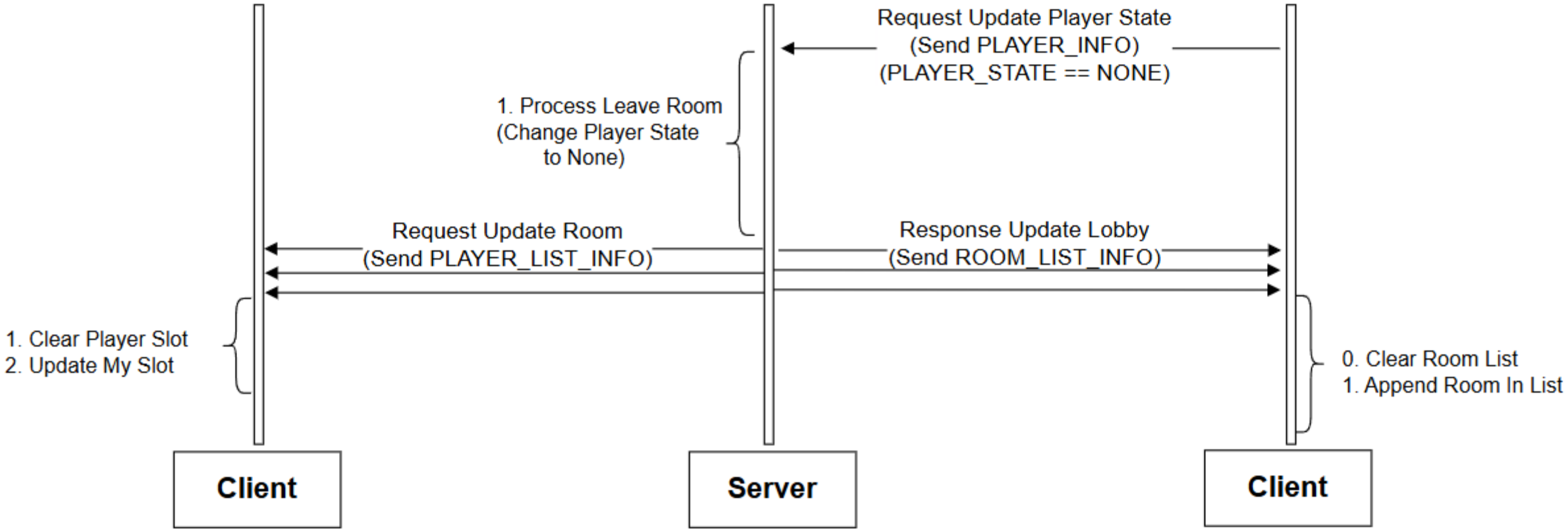
【 이벤트 다이어그램 - Room 생성 】



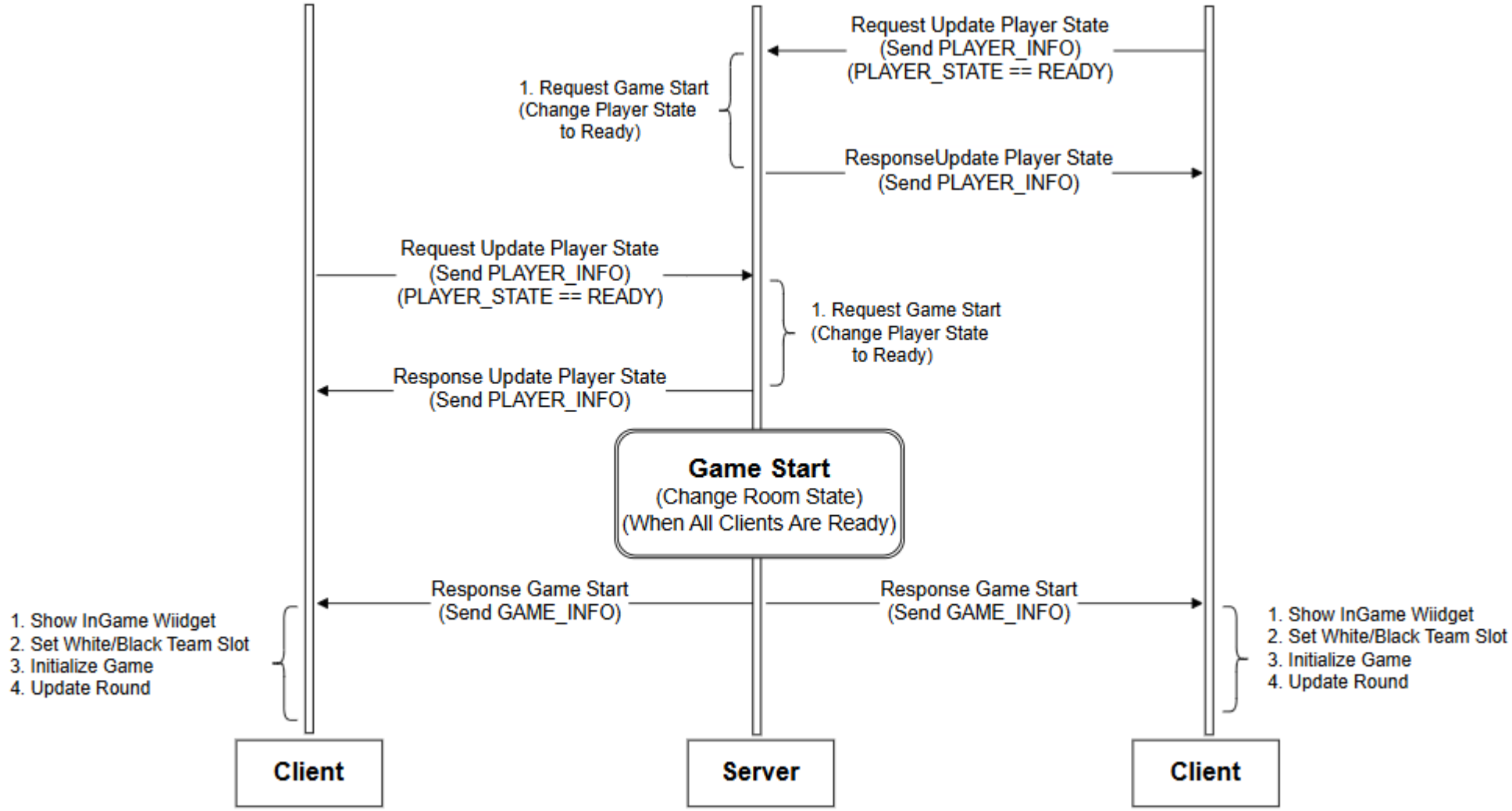
【 이벤트 다이어그램 - Room 입장 】



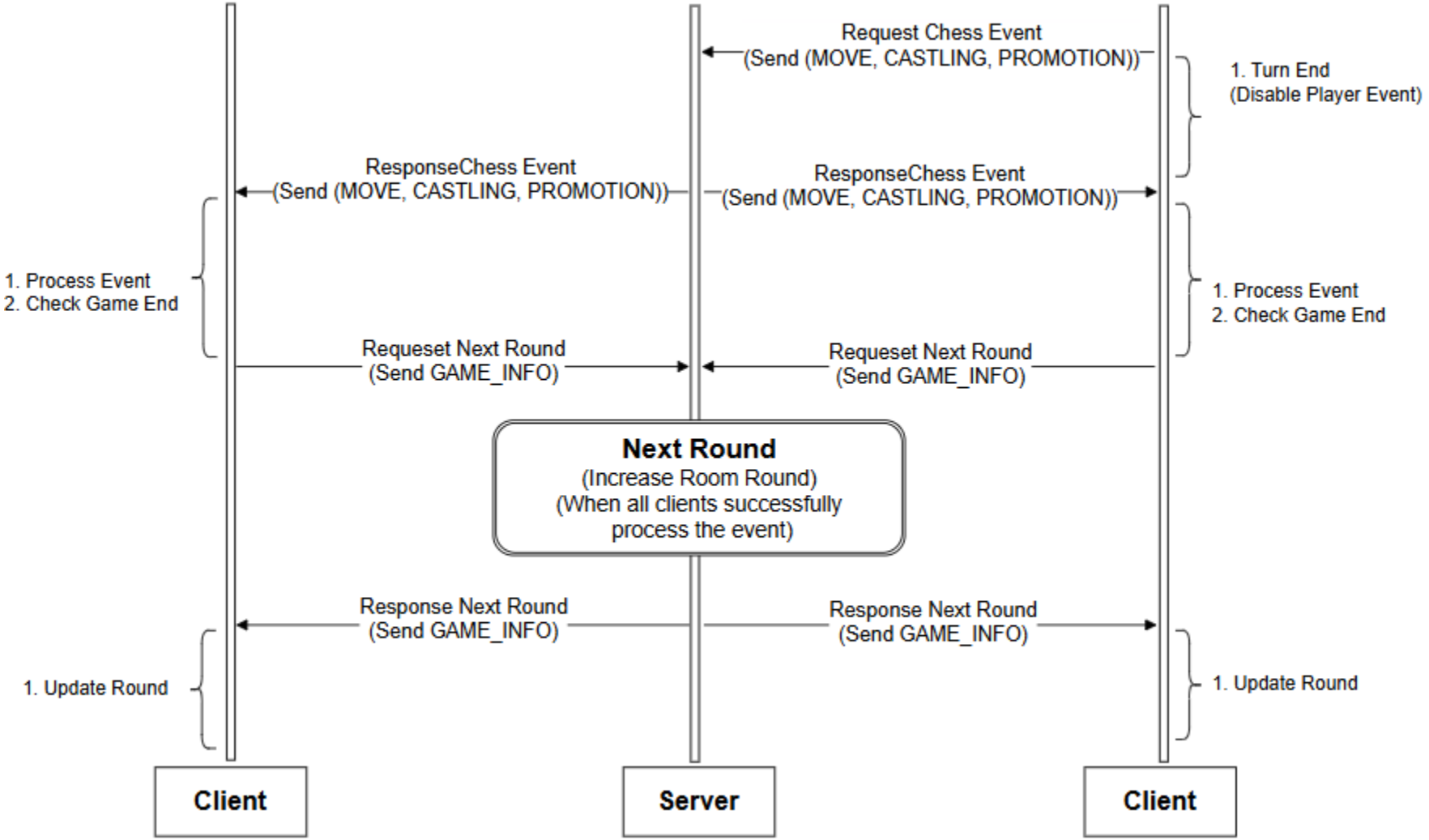
【 이벤트 다이어그램 - Room 퇴장 】



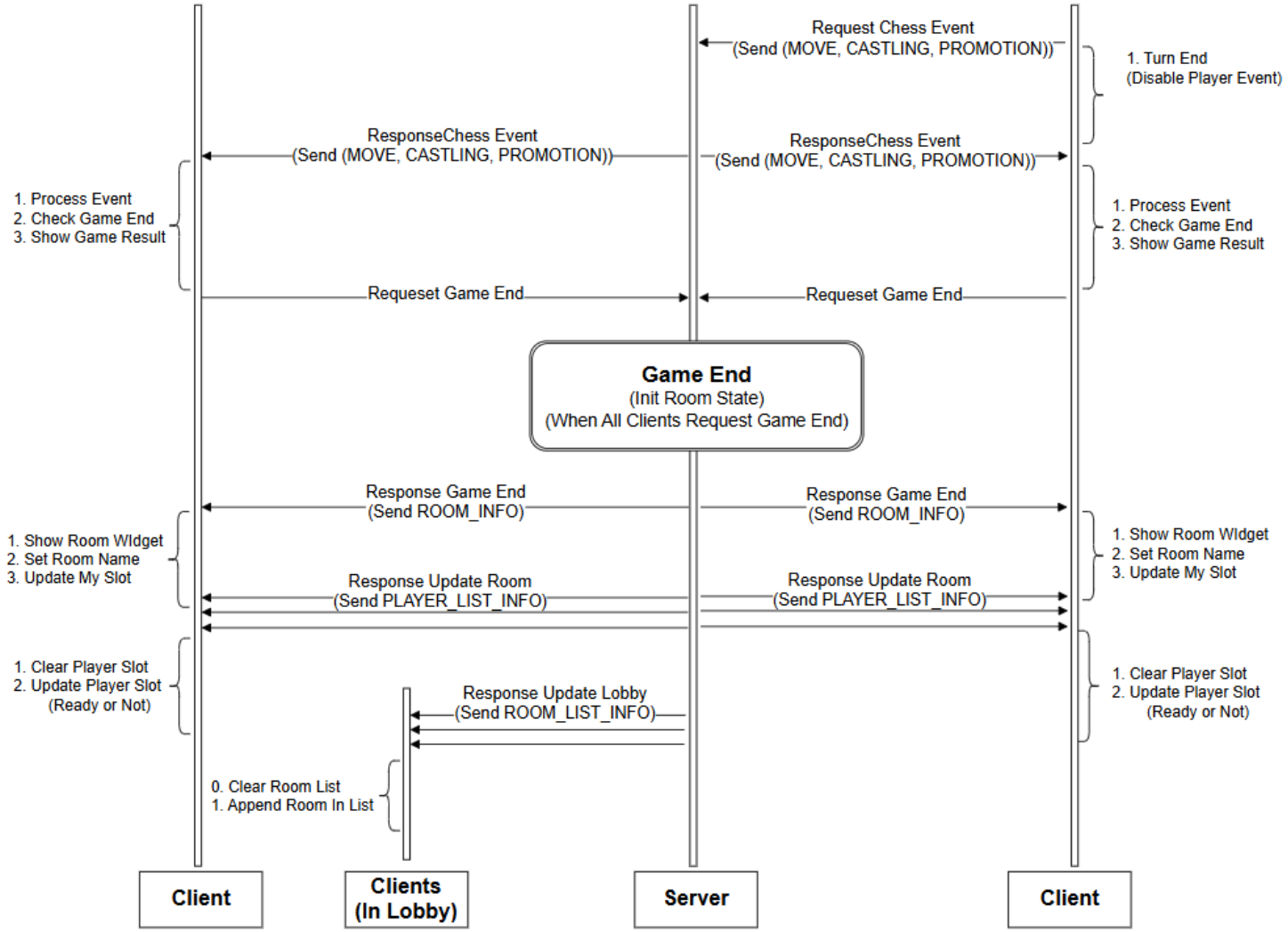
【 이벤트 다이어그램 - 게임 시작 】



【 이벤트 다이어그램 - 체스 이벤트 】



【 이벤트 다이어그램 - 게임 종료 】



【 이벤트 다이어그램 - 프로그램 종료 】