TCP/IP Socket 기반 서버-클라이언트 통신 프로그램

신선호

목차

1. 프로젝트 개요

2. Flowchart

3. UI 및 주요코드

4. 실행결과

- 본 프로젝트는 TCP/IP Socket 통신의 기본 구조를 정리하고,
  재사용 가능한 서버/클라이언트 베이스를 구축하기 위해 개발한 프로젝트입니다.

- 통신 로직을 UI와 분리하고 이벤트 기반 구조로 설계하여, 향후 다양한 프로그램에 재사용 및 확장이 가능하도록 구현하였습니다.

Server

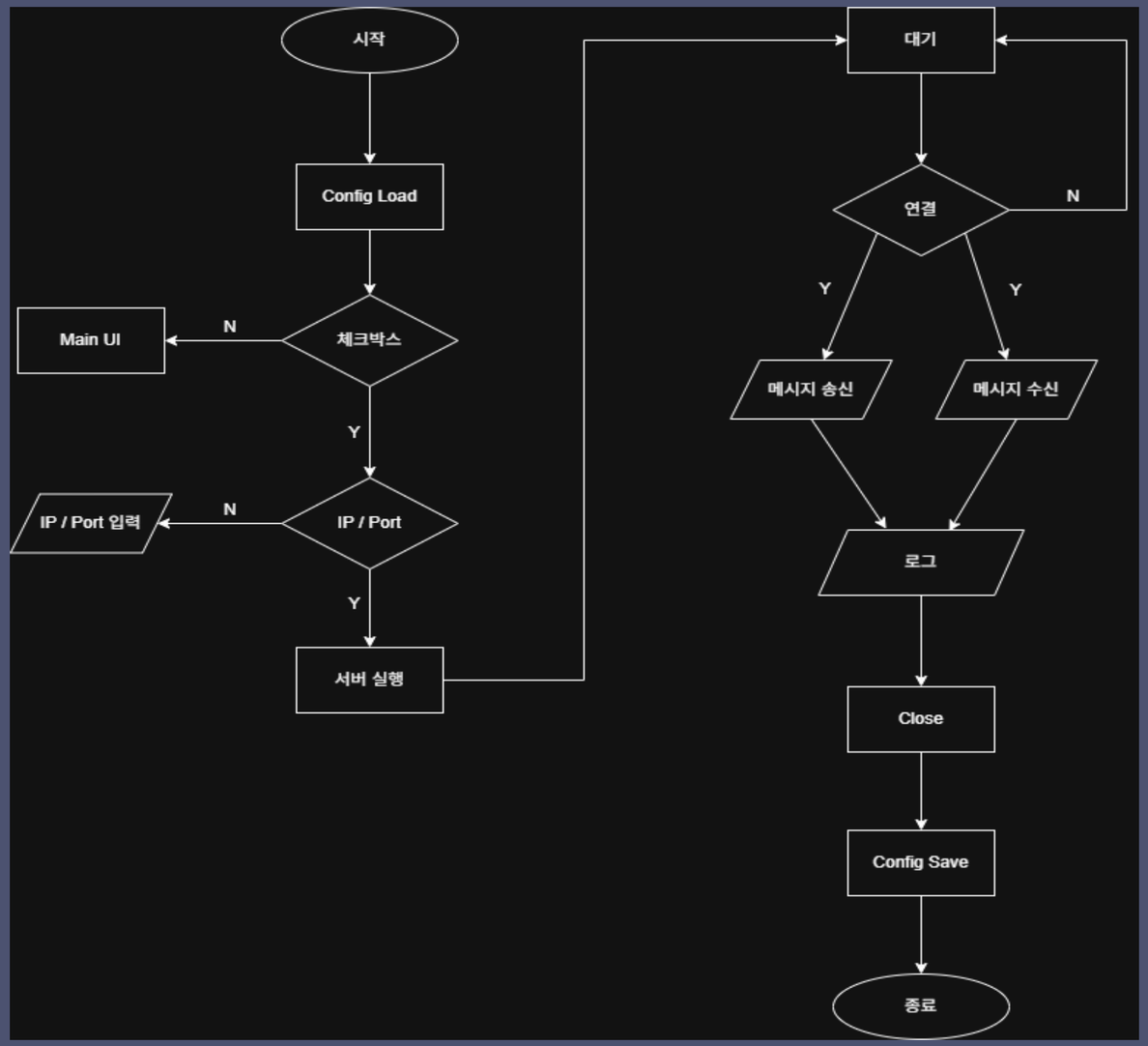
Config Load 후 설정값 검증

IP/Port 유효성 확인 및 서버 실행

연결 상태 확인 후 메시지 송신/수신 처리

송수신 데이터 로그 기록

종료 시 Config Save 후 프로그램 종료



Client

프로그램 실행 후 Config Load로 서버 설정값(IP/Port) 로드

체크박스 설정에 따라 자동 연결 여부 판단

IP/Port 유효성 검사 후 클라이언트 실행

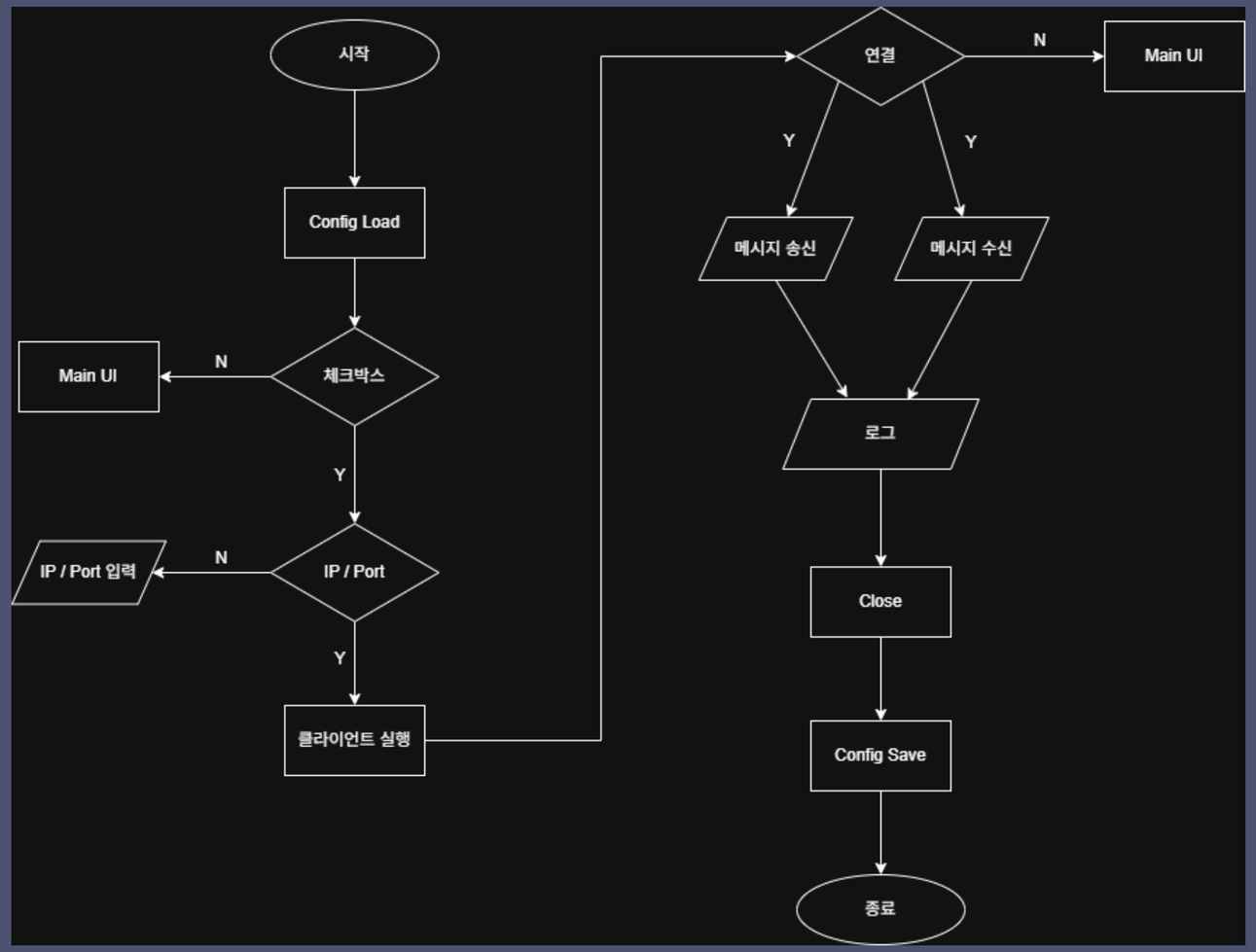
서버 연결 상태를 확인하여 연결 성공/실패 분기 처리

연결 성공 시 메시지 송신/수신 기능 수행

송수신된 메시지는 로그에 기록하여 상태 추적 가능하도록 관리

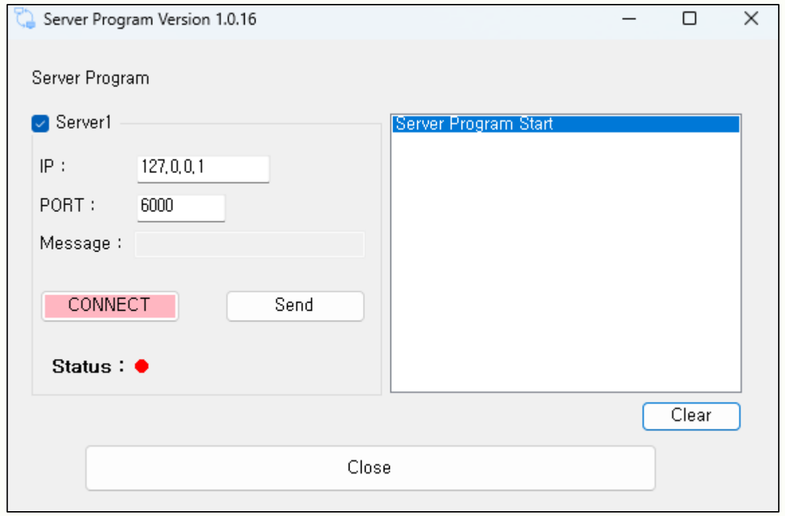
프로그램 종료 시 Close 처리 후 Config Save로 설정값 저장

저장 완료 후 프로그램 종료

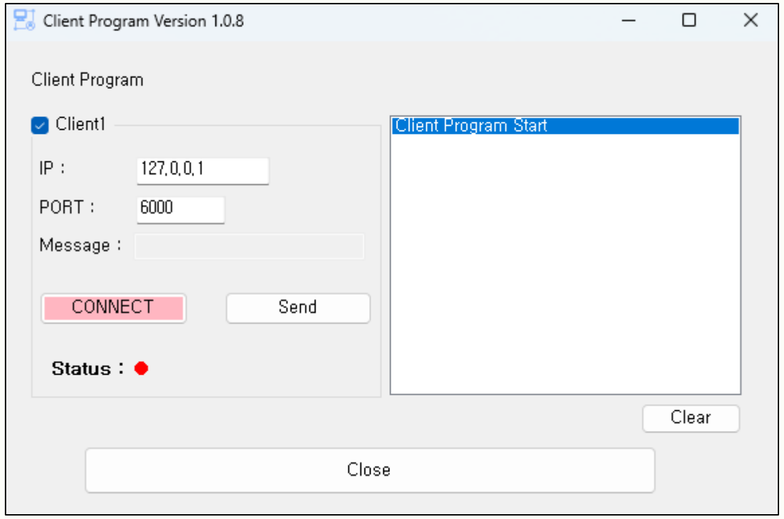


3-1. UI

Server



Client



3-2. 주요코드

[Server]

```
참조 1개
public void Connect(string ip, int port)
{
    if (IsRunning)
    {
        s_Log("이미 서버가 실행 중입니다.");

        return;
    }

    try
    {
        listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        listener.Bind(new IPEndPoint(IPAddress.Parse(ip), port));
        listener.Listen(10);

        IsRunning = true;

        listenThread = new Thread(ListenLoop);
        listenThread.IsBackground = true;
        listenThread.Start();

        s_Log($"서버 시작: {ip}: {port}");
    }
    catch (Exception ex)
    {
        s_Log($"서버 시작 실패: {ex.Message}");
    }
}
```

Socket 기반 서버를 구현하여 IP/Port 바인딩 후 Listen 상태로 전환하고,

별도 스레드에서 Accept 대기 처리하여 UI Blocking을 방지하였습니다.

3-2. 주요코드 [Server]

```
참조 1개
private void ListenLoop()
{
    while (IsRunning)
    {
        try
        {
            Socket client = listener.Accept();

            lock (lockObj)
            {
                clients.Add(client);
            }

            OnClientCountChanged?.Invoke(ConnectedClientCount);

            s_Log($"클라이언트 접속: {client.RemoteEndPoint}");

            Thread recvThread = new Thread(() => ReceiveLoop(client));
            recvThread.IsBackground = true;
            recvThread.Start();
        }
        catch
        {
            if (!IsRunning)
            {
                return;
            }
        }
    }
}
```

Accept 기반으로 클라이언트 접속을 처리하며, 접속된 클라이언트는 List에 관리하고

각 클라이언트별 수신 스레드를 생성하여 동시에 다중 통신이 가능하도록 설계하였습니다.

3-2. 주요코드 [Server]

```
참조 1개
private void ReceiveLoop(Socket client)
{
    try
    {
        byte[] buffer = new byte[4096];

        while (IsRunning && client.Connected)
        {
            int n = client.Receive(buffer);

            if (n <= 0)
            {
                break;
            }

            string msg = Encoding.UTF8.GetString(buffer, 0, n);

            s_Log($"[수신]: {msg}");
        }
    }
    catch { }
    finally
    {
        lock (lockObj)
        {
            clients.Remove(client);
        }

        if (IsRunning)
        {
            OnClientCountChanged?.Invoke(ConnectedClientCount);

            s_Log("클라이언트 연결 종료");
        }

        client.Close();
    }
}
```

클라이언트별 수신 루프를 구현하여 메시지를 실시간 처리하며,

연결 종료 시 clients 목록에서 제거하고 리소스를 정리하여 안정적으로 관리하도록 구현하였습니다.

3-2. 주요코드 [Client]

```
참조 1개
private void RunWorker()
{
    while (state != ClientState.Finished)
    {
        Thread.Sleep(10);

        switch (state)
        {
            case ClientState.Connecting:
                TryConnect();
                break;

            case ClientState.Connected:
                if (socket == null || !socket.Connected)
                {
                    state = ClientState.Disconnecting;
                }
                break;

            case ClientState.Disconnecting:
                TryDisconnect();
                break;

            case ClientState.Disconnected:
                state = ClientState.Finished;
                break;

            case ClientState.Finished:
                return;
        }
    }
}
```

Client 상태를 기반으로 Worker Thread에서 연결 흐름을 관리하여

UI Thread Blocking 없이 안정적으로 통신 상태를 제어하도록 설계했습니다.

3-2. 주요코드 [Client]

```
private void TryConnect()
{
    try
    {
        OnStatusChanged?.Invoke(ConnectionStatus.Connecting);

        c_Log($"서버({IP}:{Port}) 연결 시도 중...");

        if (socket != null)
        {
            try { socket.Close(); } catch { }
        }

        socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        var ep = new IPEndPoint(IPAddress.Parse(IP), Port);

        var result = socket.BeginConnect(ep, null, null);
        bool success = result.AsyncWaitHandle.WaitOne(2000, false);

        if (!success || !socket.Connected) { throw new Exception("연결 시간 초과 또는 서버 응답 없음"); }

        socket.EndConnect(result);
        state = ClientState.Connected;

        c_Log($"서버({IP}:{Port}) 연결됨");
        OnStatusChanged?.Invoke(ConnectionStatus.Connected);

        receiveThread = new Thread(ReceiveLoop);
        receiveThread.IsBackground = true;
        receiveThread.Start();
    }
    catch (Exception ex)
    {
        c_Log($"연결 실패: {ex.Message}");
        OnStatusChanged?.Invoke(ConnectionStatus.Error);

        state = ClientState.Disconnected;
        state = ClientState.Ready;
    }
}
```

BeginConnect + WaitOne(timeout) 방식을 적용하여

서버 연결 지연 시 무한 대기를 방지하고,

연결 성공 시 별도 수신 스레드를 생성하여

데이터 수신을 비동기로 처리했습니다.

3-2. 주요코드 [Client]

```
참조 1개
private void ReceiveLoop()
{
    try
    {
        byte[] buffer = new byte[4096];

        while (IsConnected)
        {
            int n = socket.Receive(buffer);

            if (n <= 0)
            {
                break;
            }

            string msg = Encoding.UTF8.GetString(buffer, 0, n);

            OnMessageReceived?.Invoke(msg);
        }
    }
    catch (Exception)
    {
        if (IsConnected)
        {
            c_Log("서버 연결 끊김");
        }
    }
    finally
    {
        if (IsConnected)
        {
            Disconnect();
        }
    }
}
```

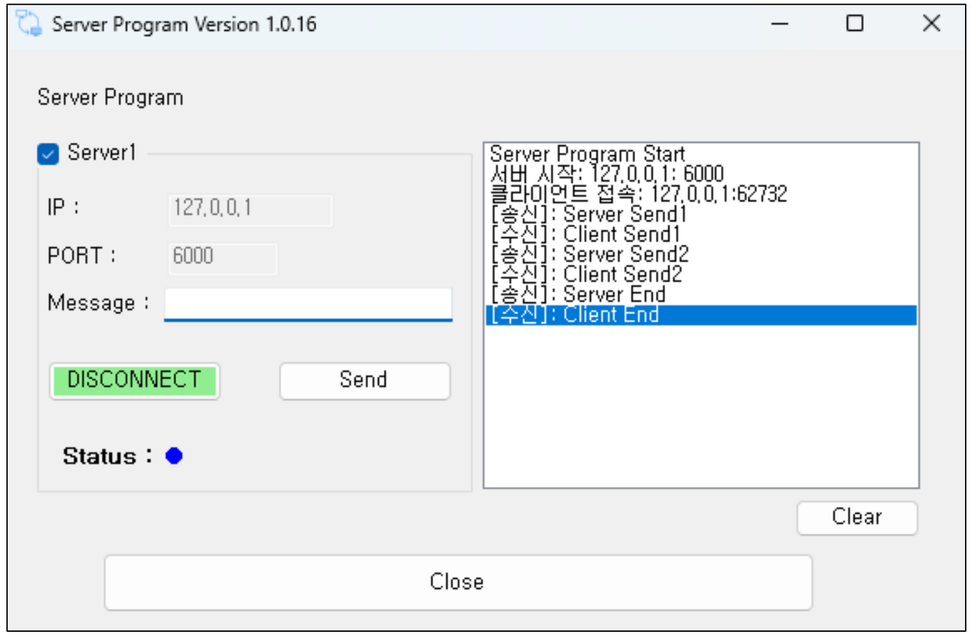
서버 수신 데이터를 ReceiveLoop에서 지속적으로 처리하고,

수신 이벤트(OnMessageReceived)를 통해 UI와 분리된 구조로 설계하여

재사용성과 유지보수성을 확보했습니다.

[Form]

Server

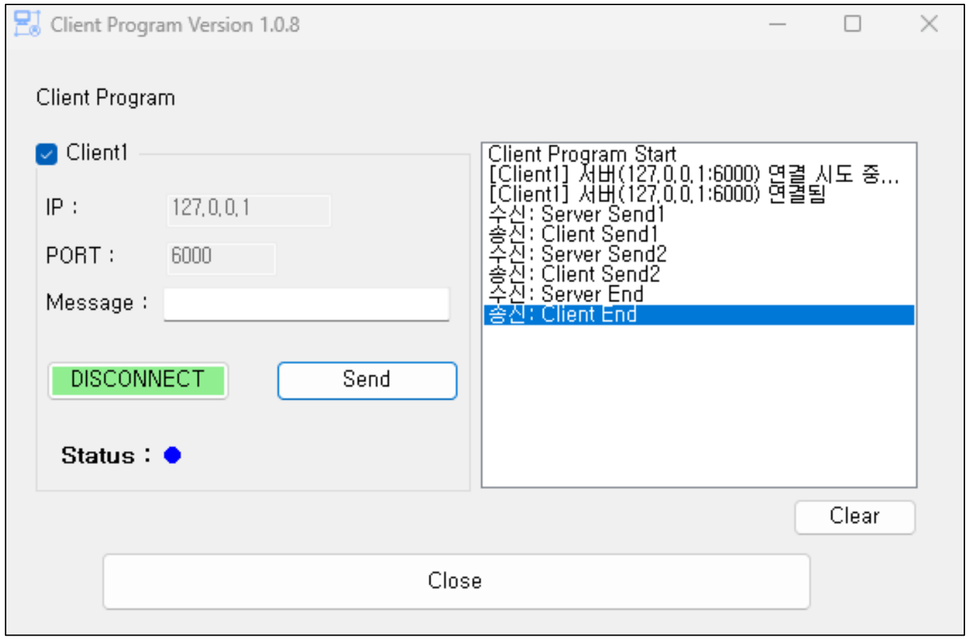


서버 실행 시 설정된 IP/Port로 소켓을 오픈하여 클라이언트 접속을 대기합니다.

클라이언트 접속/해제 상태를 실시간으로 표시하며 메시지 송수신이 가능합니다.

[Form]

Client



클라이언트 실행 시 설정된 IP/Port로 서버에 연결하여 통신을 수행합니다.

서버로부터 수신된 메시지를 UI에 표시하고,

사용자가 입력한 메시지를 송신할 수 있습니다.

[Config.INI]

Server

```
[CheckBox1]
Enable=1
[Server1]
IP1=127.0.0.1
PORT1=6000
```

Client

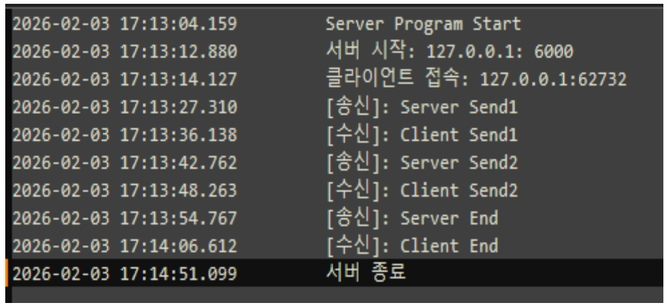
```
[CheckBox1]
Enable=1
[Client1]
IP1=127.0.0.1
PORT1=6000
```

프로그램 종료 시 입력한 IP/Port 및 체크박스 설정을 Config.ini에 자동 저장합니다.

재실행 시 Config.ini를 로드하여 이전 설정값을 자동으로 복원합니다.

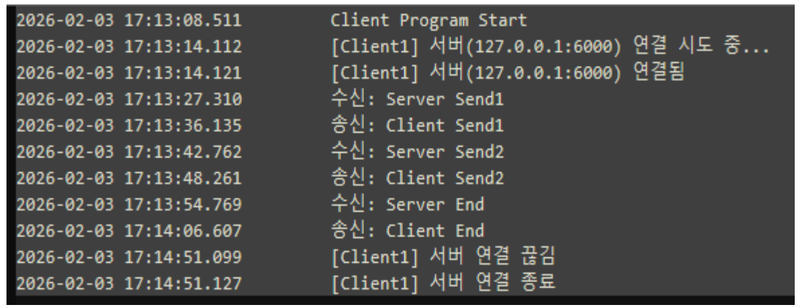
[Log]

Server



```
2026-02-03 17:13:04.159        Server Program Start
2026-02-03 17:13:12.880        서버 시작: 127.0.0.1: 6000
2026-02-03 17:13:14.127        클라이언트 접속: 127.0.0.1:62732
2026-02-03 17:13:27.310        [송신]: Server Send1
2026-02-03 17:13:36.138        [수신]: Client Send1
2026-02-03 17:13:42.762        [송신]: Server Send2
2026-02-03 17:13:48.263        [수신]: Client Send2
2026-02-03 17:13:54.767        [송신]: Server End
2026-02-03 17:14:06.612        [수신]: Client End
2026-02-03 17:14:51.099        서버 종료
```

Client



```
2026-02-03 17:13:08.511        Client Program Start
2026-02-03 17:13:14.112        [Client1] 서버(127.0.0.1:6000) 연결 시도 중...
2026-02-03 17:13:14.121        [Client1] 서버(127.0.0.1:6000) 연결됨
2026-02-03 17:13:27.310        수신: Server Send1
2026-02-03 17:13:36.135        송신: Client Send1
2026-02-03 17:13:42.762        수신: Server Send2
2026-02-03 17:13:48.261        송신: Client Send2
2026-02-03 17:13:54.769        수신: Server End
2026-02-03 17:14:06.607        송신: Client End
2026-02-03 17:14:51.099        [Client1] 서버 연결 끊김
2026-02-03 17:14:51.127        [Client1] 서버 연결 종료
```

서버/클라이언트 연결 상태 및 송수신 메시지를 로그로 기록하여 추적 가능하도록 구현했습니다.

오류 발생 시 로그 기반으로 원인 분석 및 유지보수 대응이 가능하도록 설계했습니다.

Thank You

감사합니다.