게임서버프로그래밍

2019년 2학기

한국산업기술대학교
게임공학부

정내훈

6장 게임 네트워크 엔진 프라우드넷

6.1 | 게임 서버, 네트워크 엔진

- 소켓 API만 이용해서 개발할 때의 번거러움
  1. 운영체제마다 소켓 API를 사용하는 방식이나 작동하는 방식에 조금씩 차이가 있다.
  2. 소켓 API에서 제공되지 않는 기능을 직접 만들어야 할 때가 있다.

표 6-1 게임 서버 엔진의 종류

종류	차별성	지원 운영체제	지원 언어
프라우드넷(ProudNet) http://www.proudnet.com	클라이언트-서버 간 네트워킹과 클라이언트 간 직접 네트워킹을 단순하게사용할 수 있게 한다. 네트워크 암호화, 압축, 흐름 제어 등 기능을 제공한다.	윈도, 리눅스, iOS, 안드로이드 , 플레이스테이션 4 , WebGL	C++, C#
포톤 서버(Photon Server) https://www.photonengine.com	다양한 플랫폼의 게임 클라이언트를 지원한다.	윈도, iOS , 안드로이드, 플레이스테이션, Xbox, WebGL	C#
게임스파크(GameSparks) https://www.gamesparks.com	로그인, 매치메이킹, 플레이어 정보 관리 등을 서버 개발 없이도 가능하게 한다.	구애받지 않는다 (SaaS 형태로 되어 있다).	C++, C#, Objective- C, 자바
플레이팹(PlayFab) http://www.playfab.com	게임스파크와 유사	구애받지 않는다 (SaaS 형태로 되어 있다).	C++, C#, Lua, Objective- C, 자바, 자바스크립트
락넷(RakNet) http://www.jenkinssoftware.com	유니티 엔진과 언리얼 엔진에 기본 내장되어 있는 네트워크 엔진이다.	윈도, 리눅스, iOS, 안드로이드	C++, C#

6.1 | 게임 서버, 네트워크 엔진

- 프라우드넷에 대한 내용이 나오는데 대충 훑어보고 끝낼 것임.
- 이후 내용은 시험에 안 나옴.

6.2 | 개발 환경과 기본 모듈

- 프라우드넷의 네트워크 모듈

1. NetServer 클래스: 게임 서버의 메인 모듈. 클라이언트의 연결을 받으며, 클라이언트와 메시지를 주고받는 역할을 한다. 또 각 클라이언트의 네트워크 상황 등을 열람할 수 있다.
2. NetClient 클래스: 게임 클라이언트에서 네트워크 모듈입니다. 서버로 연결을 맺은 후 메시지 주고받기를 수행할 수 있다. 또 다른 클라이언트와 P2P 통신도 가능하다.

6.3 | 게임 클라이언트-서버 간 통신

- NetServer 인스턴스를 생성하고 NetServer가 클라이언트 접속을 받으려면 다음 작업이 필요하다.
  1. CNetServer.Create()로 CNetServer 인스턴스를 생성한다.
  2. CNetServer.Start()로 서버가 클라이언트 접속을 받을 수 있게 한다.

Start()에 들어가는 필수 매개변수는 프로토콜 버전과 리스닝 포트 번호로, 프로토콜 버전은 마음대로 값을 정하면 된다.

- 클라이언트는 다음과 같이 서버에 접속한다.
  1. CNetClient.Create()로 클라이언트 인스턴스를 생성한다.
  2. CNetClient.Connect()로 서버에 접속한다.

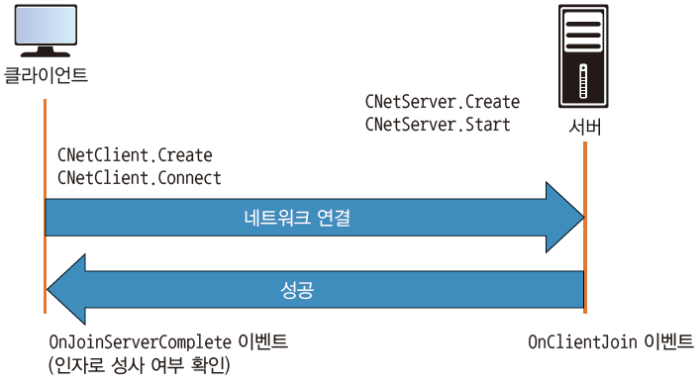

그림 6-1 클라이언트-서버 접속 과정에서 호출하는 함수와 호출되는 이벤트

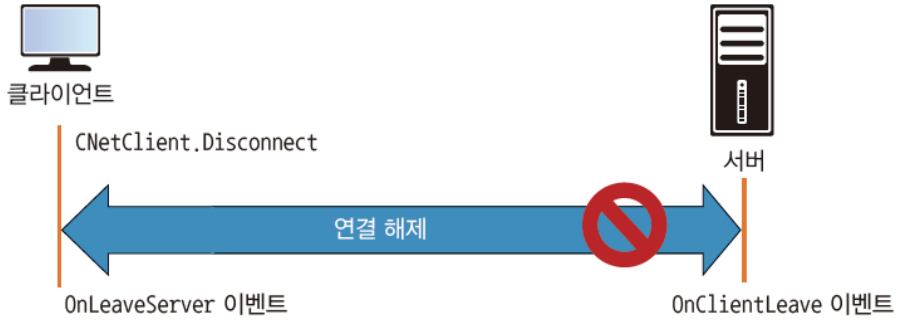

그림 6-2 클라이언트-서버 연결 해제

6.3 | 게임 클라이언트-서버 간 통신

- 클라이언트와 서버가 연결하는 과정(코드 레벨)

서버를 구동하는 코드 [C++]

```
CNetServer* s = CNetServer::Create();                    //❶
param.m_tcpPorts.Add(44444);                             //❷
param.m_udpPorts.Add(44444);
param.m_protocolVersion = Guid(
{ 0x5dca93f4, 0x8133, 0x44a0,{ 0xb5, 0x7b,
0x75, 0x7d, 0x9c, 0x78, 0xd5, 0x2e } });                 //❸
s->Start(param);
```

❶ 서버를 생성하는 과정.

❷ 서버가 네트워크 연결을 받기 위해 사용하는 TCP 리슨 포트(listen port) 값.

❸ 서버와 클라이언트가 모두 맞추어야 하는 프로토콜 버전 값.

서버를 구동하는 코드 [C#]

```
NetServer s = new NetServer(); // ❶
param.tcpPorts.Add(44444); //❷
param.udpPorts.Add(44444);
param.protocolVersion = Guid.From({ ... }); // C++ 코드의 ❸과 동일
s.Start(param);
```

6.3 | 게임 클라이언트-서버 간 통신

- 클라이언트가 서버에 접속하는 코드.

서버 접속 코드 [C++]

```
CNetClient* c = CNetClient::Create();
param.m_serverAddr = _PNT("my.game.com"); //❹
param.m_serverPort = 44444;
param.m_protocolVersion = Guid(...);            // C++ 코드의 ❸과 동일
c->Connect(param);
```

서버 접속 코드 [C#]

```
NetClient c = new NetClient();
param.serverAddr = "my.game.com";               //❹ ❹
param.serverPort = 44444;
param.protocolVersion = Guid.From({...});       // C++ 코드의 ❸ 과 동일
c.Connect(param);
```

❹ 서버에 끝점, 서버 주소와 포트 값을 넣어야 한다.

❸ 앞서 소개한 것과 똑같은 프로토콜 버전

6.3 | 게임 클라이언트-서버 간 통신

- 서버 접속이 완료되었을 때 이벤트를 처리하는 코드

이벤트 처리 코드 [C++]

```
c->OnJoinServerComplete = [](ErrorInfo* result) {        //5
        if (result->m_errorType = = ErrorType_Ok) {
                ... // 성공 처리
        }
        else {
                print(result->ToString());
                ... // 실패 처리
        }
};
```

이벤트 처리 코드 [C#]

```
c.JoinServerCompleteHandler = (result)=>{
        if (result.m_errorType = = ErrorType.Ok) {
                ... // 성공 처리
        }
        else {
                print(result.ToString());
                ... // 실패 처리
                }
};
```

6.3 | 게임 클라이언트-서버 간 통신

- 서버에서 클라이언트 접속을 받는 코드

서버에서 클라이언트 접속을 받는 코드 [C++]

```
s->OnClientJoin = [](CNetClientInfo* info) {
        // info에는 새 클라이언트 정보가 있다.
        ...
};
```

서버에서 클라이언트 접속을 받는 코드 [C#]

```
s.ClientJoinHandler = (info) {
        // info에는 새 클라이언트 정보가 있다.
        ...
};
```

6.3 | 게임 클라이언트-서버 간 통신

NetClient의 FrameMove() 함수를 호출하면, 마지막에 같은 FrameMove()를 호출했던 때 이후부터 지금까지 누적된 이벤트나 수신된 메시지에 대한 이벤트 콜백이 일어난다.

코드 [C++]

```
c->FrameMove();
```

코드 [C#]

```
c.FrameMove();
```

6.3 | 게임 클라이언트-서버 간 통신

- 클라이언트와 서버 연결이 끊어지는 처리

코드 [C++]

```
c->OnLeaveServer = [](ErrorInfo* reason) {
      // reason에는 왜 끊어졌는지가 담겨 있다.
      ...
};
```

코드 [C#]

```
c.LeaveServerHandler = (reason)=>{
      // reason에는 왜 끊어졌는지가 담겨 있다.
      ...
};
```

6.3 | 게임 클라이언트-서버 간 통신

- 서버에서도 클라이언트 연결이 끊어진 것을 확인할 수 있다.

코드 [C++]

```
s->OnClientLeave = [](CNetClientInfo* client,        //1
        ErrorInfo* reason,
        const ByteArray& comment) {
        // 여기에 이벤트 처리 코드를 추가한다.
};
```

코드 [C#]

```
s.ClientLeaveHandler = (client,reason,comment)=>{    //1
        // 여기에 이벤트 처리 코드를 추가한다.
};
```

요약

1. NetServer를 생성해서 Start() 함수를 호출한다.
2. NetClient를 생성해서 Connect() 함수를 호출한다.
3. NetServer.OnClientJoin(), OnClientLeave()에서 클라이언트의 들어옴/나감을 처리한다.
4. NetClient.OnJoinServerComplete(), OnLeaveServer()에서 서버와 접속의 성공/실패/중도 연결 해제를 처리한다.
5. NetClient.FrameMove()를 계속해서 호출한다.

6.4 | 메시지 주고받기

- 프라우드넷에서 메시지를 주고받는 방법
  1. 전통적인 방법으로 바이너리 데이터 주고받기
  2. 다른 컴퓨터에 있는 함수를 원격으로 호출하기

- 전통적인 방법으로 바이너리 데이터 주고받기

  NetClient나 NetServer의SendUserMessage() 함수를 호출하면면 상대방에게 여러분 메시지가 전송된다. 이때 제공해야 하는 매개변수는

  1. 누구한테?(HostID 또는 HostID array)
  2. 어떻게?(reliable, unreliable, …)
  3. 무엇을?(byte array)

6.4 | 메시지 주고받기

- 메시지를 보내는 루틴

코드 [C++]

```
unsigned char data[100];

c->SendUserMessage(HostID_Server,
      RmiContext::ReliableSend, data, 100);
s->SendUserMessage(ClientHostID,
      RmiContext::UnreliableSend, data, 50);

HostID sendTo[10];
      s->SendUserMessage(sendTo, 10,
            RmiContext::UnreliableSend, data, 30);
```

코드 [C#]

```
var data = new byte[100];
c.SendUserMessage(HostID.Server,
      RmiContext.ReliableSend, data);

var sendTo = new HostID[2];
s.SendUserMessage(sendTo,
      RmiContext.UnreliableSend, data);
```

6.4 | 메시지 주고받기

- 메시지를 받는 코드

코드 [C++]

```
c->OnReceiveUserMessage = [...]
	(HostID sender, const RmiContext& rmiContext,
		uint8_t* payload, int payloadLength)
	{
		... // 수신된 이벤트 처리를 해 주자.
	};
```

코드 [C#]

```
c.ReceiveUserMessageHandler =
	(sender, rmiContext, payload) => {
	{
		... // 수신된 이벤트 처리를 해 주자.
	};
```

6.5 | 와이파이 셀룰러 연결 핸드오버 기능

- 연결 유지 기능(auto connection recovery) :

  게임이 와이파이 지역을 벗어나거나 반대로 와이파이 지역 안으로 들어가더라도 연결이 끊어지는 현상 없이 게임을 플레이할 수 있다. 연결 유지 기능을 사용하려면 클라이언트에서 연결 유지 기능을 켜 두어야 한다.

코드 [C++]

```
param.m_autoConnectionRecovery = true;
...
c->Connect(param);
```

코드 [C#]

```
param.autoConnectionRecovery = true;
...
c.Connect(param);
```

6.5 | 와이파이 셀룰러 연결 핸드오버 기능

- OnServerOffline() 함수 안에서 네트워크가 일시 정지되었을 때

코드 [C++]

```
c->OnServerOffline = [...](CRemoteOfflineEventArgs &args) {
      // args에는 왜 오프라인이 되었는지에 대한 정보가 있다.
      ...
};
```

코드 [C#]

```
c.ServerOfflineHandler = (args) => {
...
}
```

6.5 | 와이파이 셀룰러 연결 핸드오버 기능

- 네트워크가 회복되면 OnServerOnline() 함수가 호출된다.

코드 [C++]

```
c->OnServerOnline = [](CRemoteOnlineEventArgs &args) {
      // args에는 연결 핸드오버가 완료된 후 정보가 담겨 있다.
...
};
```

코드 [C#]

```
c.ServerOnlineHandler = (args) = > {
...
}
```

6.6 | 원격 메서드 호출

- **RMI**

원격 메서드 호출(RemoteMethod Invocation)의 약어로, "상대방 컴퓨터 안에 있는 프로그램의 특정 함수를 멀리서 실행하라."라는 의미.

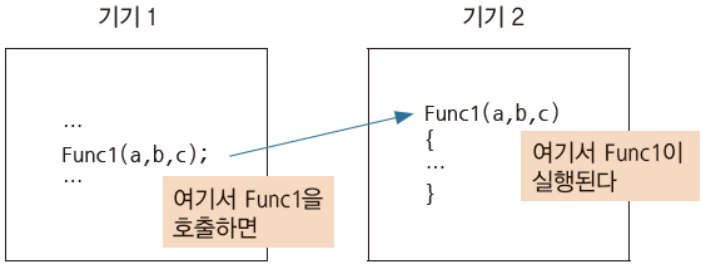

그림 6-3 프라우드넷의 원격 메서드 호출(RMI)

```
class SenderCode
{
	// 자동으로 생성되는 코드
	Knight_Move(SendTo, position, motion)
	{
		Message msg;
		msg.Write(ID_Knight_Move);
		msg.Write(position);
		msg.Write(motion);
		Send(SendTo, msg);
	}
}
```

RMI의 정체는 수동으로 만들었어야 하는 코드를 자동으로 만들어 주는 것으로 송신하는 쪽에서는 실제 함수 대신 송신을 담당하는 코드가 실행된다.

송신을 담당하는 코드는 "함수 호출을 대신 해 준다."라는 의미로 proxy라고 한다.

6.6 | 원격 메서드 호출

수신하는 쪽에서는 받은 메시지를 분석하여 여러분이 만든 함수를 호출해 준다.

```
class ReceiverCode
{
	// 자동으로 생성되는 코드
	ProcessReceivedMessage(Message msg)
	{
		ID = msg.Read();
		switch (ID)
		{
		case ID_Knight_Move:
				position = msg.Read();
				motion = msg.Read();
				Knight_Move(position, motion);
				break;
				...
		}
	}

	Knight_Move(position, motion)
	{
		// 여기에 코드를 작성한다.
	}
}
```

수신을 담당하는 코드는
"여러분 함수를 호출해 주는 기반"이라는
의미로 stub라고 한다.

6.6 | 원격 메서드 호출

- 원격 메서드 호출의 장점

1. 송신을 처리하는 코드와 수신을 처리하는 코드를 손으로 일일이 구현할 필요가 없다.
2. 송수신 메시지의 형태가 변경되었을 때 송신이나 수신을 처리하는 코드를 수정하다가 실수할 위험이 없다.

- 원격 메서드 호출을 사용해서 "세상에서 제일 비효율적인 계산기" 만들기

RMI 함수 선언하기_RMI 함수 선언은 PIDL 확장자를 가진 파일에서 한다.

1. 클라이언트는 서버에 값 2개를 보낸다.
2. 서버는 값 2개를 받아서 합친 값을 클라이언트에 보낸다.

```
global CalcC2S 1000
{
	RequestAdd([in] int a, [in] int b);
}

global CalcS2C 2000
{
	ResponseAdd([in] int sum);
}
```

6.6 | 원격 메서드 호출

RMI ID : 여러분이 선언하는 RMI 함수 각각은 서로 다른 메시지 ID를 가져야 한다.
이 코드의 1000, 2000은 RMI ID의 시작 값.
여러분이 선언하는 각각의 RMI 함수는 서로 다른 RMI ID를 가진다.
선언된 순서대로 1 큰 수를 가진다.

```
global CalcC2S 1000
{
	RequestAdd([in] int a, [in] int b);
}

global CalcS2C 2000
{
	ResponseAdd([in] int sum);
}
```

각각의 RMI 는 자동으로 1001,1002,1003을 가진다.

6.6 | 원격 메서드 호출


```
global CalcC2S 1000
{
	RequestAdd([in] int a, [in] int b);
}

global CalcS2C 2000
{
	ResponseAdd([in] int sum);
}
```

PIDL 파일로 컴파일하면 송신을 담당하는 코드,
즉 Proxy와 수신을 담당하는 코드인 Stub이
만들어진다.
각각 클래스 형태.

```
// 클라이언트 → 서버 보내는 측
CalcC2S::Proxy
// 클라이언트 → 서버 받는 측
CalcC2S::Stub
CalcC2S::StubFunctional

// 서버 → 클라이언트 보내는 측
CalcS2C::Proxy
// 서버 → 클라이언트 받는 측
CalcS2C::Stub
CalcS2C::StubFunctional
```

PIDL 파일을 컴파일할 때 PIDL 컴파일러가
생성하는 클래스

6.6 | 원격 메서드 호출


```
lass MyStub: public CalcC2S:Stub
{
	RequestAdd(from, rmiContext, a, b)
	{
		// 여기에 코드를 입력한다.
	}
}

myStub = new MyStub;
```

Stub을 사용할 때는 다음과 같이 상속 클래스를 만든다.

```
myStub = new CalcC2S::StubFunctional();

myStub.RequestAdd = [](from, rmiContext, a, b)
{
	// 여기에 코드를 입력한다.
}
```

StubFunctional() 함수를 쓸 때는 클래스를 그대로 사용한다. 여러분이 정의한 RMI 함수는 함수 객체 타입의 변수이므로 각 함수 객체에 루틴, 즉 함수나 람다식을 넣으면 된다.(모던 C++ 방식).

6.6 | 원격 메서드 호출

- 서버와 클라이언트에 송신 코드(Proxy)와 수신 코드(Stub)를 붙이는 방법

  1. Proxy, Stub이 인스턴스를 생성한다.
  2. 이 인스턴스들을 AttachProxy, AttachStub 함수를 써서 부착한다.

코드 [C++]

```
CalcC2S::Proxy CalcC2SProxy;                    //❶
c->AttachProxy(&CalcC2SProxy);                  //❷
CalcS2C::StubFunctional CalcS2CStub;            //❶
c->AttachStub(&CalcS2CStub);                    //❷
```

코드 [C#]

```
CalcC2S.Proxy CalcC2SProxy;                 //❶
c.AttachProxy(CalcC2SProxy);                    //❷
CalcS2C.StubFunctional CalcS2CStub;             //❶
c.AttachStub(CalcS2CStub);                  //❷
```

6.6 | 원격 메서드 호출

- 클라이언트에서 서버로 RMI를 호출하기.

RMI를 호출할 때는 다음 매개변수를 넣는다.
첫 번째 매개변수: 수신자 HostID 1개 혹은 배열
두 번째 매개변수: RmiContext
나머지: 여러분이 정의한 RMI 함수의 매개변수

코드 [C++]

```
CalcC2SProxy.RequestAdd(HostID_Server,
        RmiContext::ReliableSend,
        3, 4);
```

코드 [C#]

```
CalcC2SProxy.RequestAdd(HostID.Server,
        RmiContext.ReliableSend,
        3, 4);
```

6.6 | 원격 메서드 호출

코드 [C++]

```
CalcC2SStub.RequestAdd_Function =
        []PARAM_CalcC2SStub_RequestAdd
        { //❶
                int sum = a + b; //❷
                CalcS2CProxy.ResponseAdd(remote,
                RmiContext::ReliableSend,
                sum);
        }; //❸
};
```

코드 [C#]

```
CalcC2SStub.RequestAdd =
        (remote, rmiContext, a, b)
        { //❶
                int sum = a + b; //❷
                CalcS2CProxy.ResponseAdd(remote,
                        RmiContext.ReliableSend,
                        sum); //❸
        };
};
```

- 서버에서 RMI 호출받기

호출받는 함수에서 전달하는 매개변수
첫 번째 매개변수: 송신자 HostID
두 번째 매개변수: RmiContext
나머지: 여러분이 정의한 매개변수

❶ 클라이언트에서 메시지를 보낸, 즉 원격으로 호출한 RequestAdd() 함수가 실행할 루틴을 람다식으로 넣어 주는 부분

❷ 두 값을 더한다.

❸ 서버에서 클라이언트에 원격으로 함수를 호출한다.
즉, 서버에서 클라이언트로 두 값을 더한 결과 메시지를 보냅니다. 여기서 remote는 앞서 수신할 때 송신자의 식별자(HostID)이다.

6.6 | 원격 메서드 호출

- 서버에서 클라이언트로 원격으로 함수를 호출하려면

  앞서 선언했던 <서버=>클라이언트> RMI, 즉 CalcS2C를 서버와 클라이언트에 붙여야 한다.

  CalcS2C의 Proxy는 서버에 붙이고 CalcS2C의 Stub은 클라이언트에 붙인다.

코드 [C++]

```
CalcS2CStub.ResponseAdd_Function =
	[]PARAM_CalcS2CStub_ResponseAdd
	{
		print(sum);
	};
```

코드 [C#]

```
CalcS2CStub.ResponseAdd =
(remote, rmiContext, sum)
{
	print(sum);
};
```

6.6 | 원격 메서드 호출

- 지금까지 절차 요약

1. PIDL 파일에 RMI 함수들을 정의한다.
2. 이를 컴파일한다.. 가능하면 빌드 설정에 넣는다.
3. 생성된 Proxy, Stub을 NetClient와 NetServer에 부착한다.
4. 생성된 Proxy의 함수를 호출하면 메시지가 전송된다.
5. 생성된 Stub에 함수를 부착하면 그 함수들이 호출된다.

6.7 | 클라이언트끼리 P2P 통신

- P2P(peer-to-peer)

1. 서버에서 클라이언트 1과 클라이언트 2가 P2P 연결을 하라고 지시한다.
2. 클라이언트 1, 클라이언트 2는 자기가 P2P 연결이 되었음을 즉시 알 수 있다.
3. 직후에 바로 클라이언트 1과 클라이언트 2는 서로 메시지를 주고받는다.

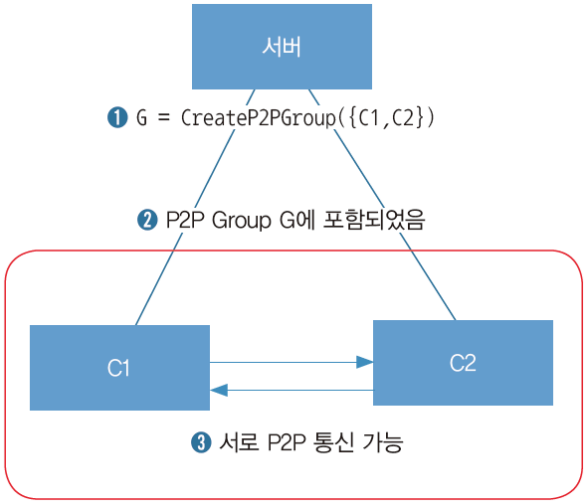

그림 6-4 P2P 연결을 위한 P2P 그룹

- P2P 그룹의 특징

P2P 그룹에는 클라이언트를 0개 이상 넣을 수 있다.

클라이언트 하나가 여러 P2P 그룹에 들어가도 된다. (겹쳐도 된다.)

서버도 P2P 그룹에 들어가는 것이 허락된다.

6.7 | 클라이언트끼리 P2P 통신

- P2P 그룹 만들기 과정

코드 [C++]

```
HostID list[2];
list[0] = C1;
list[1] = C2;
// 두 번째 인자 = 배열 크기
G = s->CreateP2PGroup(list, 2);
```

코드 [C#]

```
G = s.CreateP2PGroup({C1, C2});
```

6.7 | 클라이언트끼리 P2P 통신

- 클라이언트는 G의 존재를 어떻게 알까?

코드 [C++]

```
c->OnP2PMemberJoin = [...]
        (HostID memberHostID,          //1
        HostID groupHostID,
        //2
        int memberCount,               //3
        const ByteArray &customField)
{
        G = groupHostID;               //4
        Peers.Add(memberHostID);
}
```

코드 [C#]

```
c.P2PMemberJoinHandler =
(memberHostID,                  //
1
groupHostID,                // 2
memberCount,                    //
3
customField)=>{
G = groupHostID;            // 4
Peers.Add(memberHostID);
```

1. memberHostID가 가리키는 다른 호스트와 로컬 자기 자신과 P2P 연결이 맺어졌는지 의미한다.
2. memberHostID와 로컬 자기 자신이 어느 P2P 그룹에 들어가 있는지 의미한다.
3. 이제 P2P 그룹에 호스트가 몇 개 들어가 있는지 의미한다.
4. 여러분 코드. 지금 일단 필요한 것은 "내가 어느 P2P 그룹에 있고 나와 통신 가능한 다른 호스트들이 누구인지"를 보관하는 것이다.

6.7 | 클라이언트끼리 P2P 통신

- P2P 메시지 보내기

코드 [C++]

```
c->SendUsermessage(G, RmiContext::ReliableSend, data, length);
```

코드 [C#]

```
c.SendUserMessage(G, RmiContext.ReliableSend, data);
```

6.7 | 클라이언트끼리 P2P 통신

- P2P 그룹에 멤버 추가하기/삭제하기/파괴하기

  이미 만들어진 P2P 그룹에 JoinP2PGroup을 호출해서 더 많은 클라이언트를 기존 P2P 그룹에 넣을 수 있다.

  그러면 새로 추가된 호스트와 기존에 있던 호스트는 신규로 들어오는 것에 대한 OnP2PMemberJoin() 이벤트를 받는다.

  P2P 그룹에서 호스트를 제거하려면 NetServer.LeaveP2PGroup()을 호출하고, P2P 그룹 자체를 파괴하려면 DestroyP2PGroup()을 호출

6.7 | 클라이언트끼리 P2P 통신

- 홀펀칭 상태 변화 감지하기

P2P 홀펀칭 : 인터넷 공유기 뒤에 있는 클라이언트끼리도 서로 P2P 통신을 할 수 있는 기법

홀펀칭이 성공하면 클라이언트에서는 이를 통지받는다. (OnChangeP2PRelayState 이벤트.)

반대로 기존에 있던 홀펀칭이 중도에 사라지는 경우에도 통지를 받는다.

코드 [C++]

```
c->OnChangeP2PRelayState = [...]
      (HostID remoteHostID, ErrorType reason)
//❶❶
{
      ...
}
```

코드 [C#]

```
c.ChangeP2PRelayStateHandler =
      remoteHostID, reason)=>{ // ❶
...
};
```

❶ 어떤 상대방 클라이언트에 대한 홀펀칭 상태가 바뀌었는지, 바뀐 상태가 어떤지 알려줌.reason = Ok면 홀펀칭이 되어 있다는 의미이다. 다른 값이면 홀펀칭이 중도 사라졌음을 의미하며, 왜 사라졌는지 보여 준다.

6.7 | 클라이언트끼리 P2P 통신

- P2P 통신에서도 RMI를 사용할 수 있다.

  P2P RMI 함수를 선언했다면 Proxy와 Stub의 클래스 인스턴스를 여러분 클라이언트에 모두 붙여야 한다.

```
[MyGame.pidl]

global MyGameP2P //❶
{
    Player_Move([in] Vector3 position);
}

[C++]
MyGameP2P::Proxy P2PProxy; //❷
MyGameP2P::StubFunctional P2PStub;

P2PStub.Player_Move_Function = [...]PARAM_MyGameP2P_Player_Move{ //❸
    ...
};

c->AttachProxy(&P2PProxy); //❹
c->AttachStub(&P2PStub);

P2PProxy.Player_Move(G, RmiContext::UnreliableSend, myPosition); //❺
```

6.7 | 클라이언트끼리 P2P 통신

코드 [C#]

```
MyGameP2P.Proxy P2PProxy; //❷
MyGameP2P.Stub P2PStub;
P2PStub.Player_Move = (sendFrom, rmiContext, position)=>{ //❸
...
};

c.AttachProxy(P2PProxy); //❹
c.AttachStub(P2PStub);

P2PProxy.Player_Move(G, RmiContext.UnreliableSend, myPosition); //❺
```

❶ P2P 통신용 RMI를 정의한다.

❷ P2P RMI의 Proxy와 Stub의 클래스 인스턴스입니다

❸ P2P로 RMI를 받으면 그것을 처리하는 함수를 정의합니다. (여러분 마음대로 내용을 채우세요)

❹ ❷ 의 인스턴스들을 NetClient에 부착한다.

❺ P2P 클라이언트에 원격 함수 호출, 즉 송신을 한다.

6.7 | 클라이언트끼리 P2P 통신

- 요약하기

CreateP2PGroup()으로 클라이언트들이 직접 통신할 P2P 그룹을 만든다.

OnP2PMemberJoin()으로 클라이언트들은 자기가 P2P 통신할 수 있음을 안다.

RMI나 SendUserMessage()로 P2P 메시지를 보낸다.

P2P RMI를 쓰려면 Proxy, Stub을 모두 NetClient에 부착해야 한다.

6.8 | 예시 : 채팅 처리

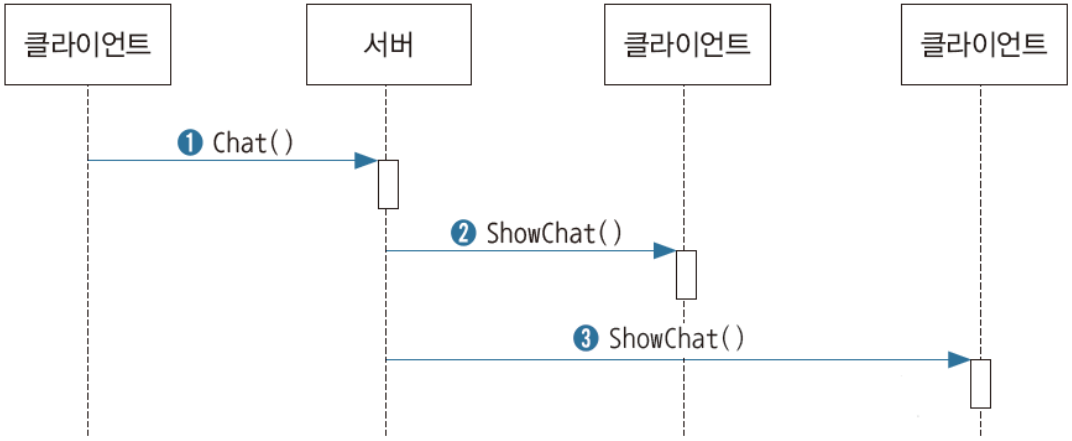

그림 6-5 채팅 프로그램 예

1. 클라이언트에서 서버로 보내는 Chat RMI 함수를 선언한다. 클라이언트에서는 이 함수를 호출하여 서버에 채팅을 보낸다.
2. 서버에서는 이를 수신하면 다른 모든 클라이언트에 ShowChat RMI를 멀티캐스트 한다.
3. 클라이언트는 ShowChat RMI를 받으면 이를 화면에 표시한다.

6.8 | 예시 : 채팅 처리


```
class MyGameServer
{
	CNetServer* m_netServer;
	CriticalSection m_critSec;
	map<HostID, shared_ptr<RemoteClient> >
		m_remoteClients;
	OnClientJoin(clientInfo)
	{
		CriticalSectionLock lock(m_critSec, true);
		shared_ptr<RemoteClient> newRemote =
			shared_ptr<RemoteClient>(new RemoteClient);
		fill_something(newRemote);
		m_remoteClients.Add(clientInfo->m_hostID,
		newRemote);
	}

	OnClientLeave(clientInfo)
	{
		CriticalSectionLock lock(m_critSec, true);
		m_remoteClients.Remove(clientInfo->m_hostID);
```

6.8 | 예시 : 채팅 처리


```
        }

        Init()
        {
                m_netServer->OnClientJoin = OnClientJoin;
                m_netServer->OnClientLeave = OnClientLeave;
        }
}
```

MyGameServer 서버 메인 클래스를 만든다.

NetServer에서는 RMI나 이벤트 함수 호출이 여러 스레드에서 실행된다.

OnClientJoin() 함수에서는 새 클라이언트가 들어옴을 안다.

반대로 OnClientLeave() 함수에서는 나간 클라이언트에 대한 데이터를 삭제한다.

NetServer는 멀티스레드로 작동하도록 기본 설정되어 있다. 그렇지만 원한다면 싱글스레드로 작동하도록 설정할 수도 있다.

NetServer가 멀티스레드로 작동하면 같은 클라이언트에서 수신은 서로 다른 스레드에서 호출되지 않게 제어한다.

6.8 | 예시 : 채팅 처리

- 클라이언트에서 서버와 연결 및 수신 처리

1. NetClient.FrameMove()를 호출한다. 이때 이벤트 및 수신 처리를 하게 된다.
2. 서버와 연결하는 과정이 성공하거나 실패하면 OnJoinServerComplete()가 호출된다. 이 함수를 구현한다.
3. 서버와 연결이 중도 해제될 경우 OnLeaveServer() 콜백이 발생한다. 역시 이 함수도 구현한다.

```
class MyGameClient
{
	CNetClient* m_netClient;

	OnJoinServerComplete(info, replyFromServer)
	{
		if (info.type = = OK)
		{
			do_success();
		}
		else
		{
			do_failure();
		}
	}
```

6.8 | 예시 : 채팅 처리


```
        OnLeaveServer(info)
        {
                do_leave();
        }
        Init()
        {
                m_netClient->OnJoinServerComplete =
                OnJoinServerComplete;
                m_netClient->OnLeaveServer = OnLeaveServer;
        }

        MainLoop()
        {
                while (true)
                {
                        m_netClient->FrameMove();
                        update_scene();
                        render_scene();
                }
        }
}
```

6.8 | 예시 : 채팅 처리

- 서버에서 메시지를 수신하면 일을 처리하는 코드는 다음과 같이 개발한다.

```
class RemoteClient
{
	string m_name;
};

MyGameServer:public MyGameC2S::Stub
{
	MyGameS2C::Proxy m_s2cProxy;

	MyGameC2S::Stub::Chat(senderHostID, rmiContext, text) // [**]
	{
		CriticalSectionLock lock(m_critSec, true);

		// 송신자 정보 가져오기
		shared_ptr<RemoteClient> sender =
			m_remoteClients.find(senderHostID).second;
```

6.8 | 예시 : 채팅 처리


```
            // ❶수신자 목록 만들기
            vector<HostID> sendTo;
            for (auto r : m_remoteClients)
            {
                  if (r.first != senderHostID)
                        sendTo.push_back(r.first);
            }

            // ❷멀티캐스트!
            m_s2cProxy.ShowChat(&r[0], r.size(),
                  sender->m_name,
                  text);
      }
}
```

[**] RMI를 이용해서 채팅 메시지를 받는다.

❶ 채팅 메시지를 받았으니 채팅받을 클라이언트들을 모은다.

❷ 이렇게 모은 후에는 클라이언트 목록을 RMI의 매개변수로 넣어서 송신한다. 이렇게 하면 여러 클라이언트에 원격으로 함수가 호출된다. (멀티캐스트를 한다.)

6.9 | 스레드 모델

그림 6-6 프로세스 하나가 여러 스레드를 가질 때와 여러 프로세스가 각각 스레드 하나를 가질 때

프라우드넷은 서버 프로세스 하나가 스레드를 여럿 가지는 형태의 서버와 여러 서버 프로세스가 각각 스레드 하나를 가지는 형태의 서버에 대한 스레드 모델을 모두 지원한다.

기본값으로 NetServer의 워커 스레드(worker thread)는 CPU 개수만큼 구동됩니다. 즉,CPU 개수만큼 스레드를 가진 스레드 풀이 기본으로 제공된다.

그러나 다음 모든 상황일 때는 이를 권장하지 않는다.

1. 서버 내부 데이터가 뮤텍스 1개로 보호되는 경우
2. 서버 내부 로직에서 DB나 파일 액세스 등 디바이스 타임(device time)이 없는 경우
3. 서버 프로세스를 여러 개 띄울 수 있게 분산 서버로 개발하는 경우

6.9 | 스레드 모델

- NetServer를 싱글스레드로 구동할 때

```
void main()
{
	...
	CThreadPool* p = CThreadPool::Create(..., 0); //❶
	
	CStartServerParameter param;
	...
	param.m_externalNetWorkerThreadPool = p; //❷
	param.m_externalUserWorkerThreadPool = p;
	netServer->Start(param);

	while (true)
	{
		...
		//❸ 최대 10밀리초까지 기다리면서,
		// thread pool에 쌓인 이벤트를 처리한다.
		p->Process(10);
	}
}
```

❶ 스레드가 전혀 없는 스레드 풀 객체를 생성.

❷ NetServer가 이 스레드 풀 객체를 사용하도록 설정한다.

❸ 수동으로 스레드 풀을 숨쉬게(?) 하는 함수를 지속적으로 호출한다.

6.10 | 더 읽을 거리

http://guide.nettention.com은 프라우드넷 도움말로 자세한 사용법이 나와 있다. 여기에는 튜토리얼도 있다.

프라우드넷에 있는 채팅 예제 소스(ProudNet/Samples/Chat) 등 여러 샘플과 도움말을 참고.

도움말에는 유니티를 혼용하여 게임을 만드는 동영상도 있다.

유니티에서 캐릭터 이동 동기화에 대한 튜토리얼은 https://github.com/Nettention/CharacterMove에서 볼 수 있다.

추측항법과 가시 영역 필터링 혼용 예제(ProudNet/Sample/SynchWorld)에서는 실시간 멀티플레이와 MMO 게임 서버에서 대량의 플레이어 처리를 하는 데 필요한 지식을 얻을 수 있다.

강사의 평가

좋은 엔진이고, 생산성도 높고, 성능도 좋고, 가격도 싸다.

DirectX를 쓰지 않고 Unity를 사용하는 것과 비슷하다.

MMO서버를 프라우드 넷으로 만드는 경우는 거의 없다. (성능 문제가 아니라 서버 프로그래머의 자존심???)

- 서버 제작 시 가장 큰 문제는 멀티쓰레드 문제인데 그 문제를 해결해 주지 않는다.