

유니티 멀티플레이어 환경에서의 경량 암호 통신 구현

지도교수 : 손준영

팀명 : 민트초코칩시

팀원 : 201924409 고세화

201924438 김재민

201924521 이강인

목차

1 요구조건 및 제약 사항 분석에 대한 수정 사항

- 1.1 요구조건
- 1.2 제약 사항 분석
- 1.3 수정 사항

2 설계 상세화

- 2.1 블록 암호 알고리즘 및 운영모드 설계
- 2.2 공개키 암호 알고리즘 및 서명 설계
- 2.3 통신 프로토콜 설계
- 2.4 게임 서버 설계
- 2.5 게임 클라이언트 설계

3 구성원별 진척도

4 보고 시점까지의 과제 수행 내용 및 중간 결과

- 4.1 블록 암호 알고리즘 및 운영모드 구현
- 4.2 공개키 암호 알고리즘 및 서명 구현
- 4.3 게임 서버 구현
- 4.4 게임 클라이언트 구현

5 멘토의견서 피드백 반영사항

1 요구조건 및 제약 사항 분석에 대한 수정사항

1.1 요구조건

본 과제는 온라인 게임에서 안전한 통신 환경을 만드는 것이 첫 번째 목표이며, 블록 암호 알고리즘별 성능 비교 분석이 두 번째 목표이다. 이에 따른 요구 조건은 다음과 같다.

- 최초 세션 수립 시 서버 인증서 확인 및 키 교환에 필요한 공개키 암호 알고리즘 구현
- 성능 비교를 위한 블록 암호 알고리즘 및 운영 모드 구현
- 각 암호 알고리즘을 적용 가능한 서버/클라이언트 구현

또한, 유니티와의 이식성을 위해 모든 코드는 C#으로 작성하였다.

1.2 제약 사항 분석

본 과제는 게임 네트워크 보안에 초점을 두며, 다음과 같은 제약 사항을 가진다.

- 온라인 게임 보안에서 필요한 안티 치팅, 안티 디버깅, 코드 난독화 등의 클라이언트 보안 내용은 과제 범위에서 제외된다. 이는 주어진 과제인 네트워크 보안에 집중하기 위함이다.
- 모든 게임 패킷을 암호화하며, 이는 실제 온라인 게임이 동작하는 것과 상이할 수 있다.
- 최초 세션 수립 시 세션키 교환을 하게 된다. 이때, 표준 TLS 프로토콜을 참고해 구현하지만, 온라인 게임에 맞게 과정을 단순화하며, OpenSSL 을 통한 테스트가 불가하다.
- 각 블록 암호 알고리즘별 성능 비교는 암호/복호화 시간만 측정하여 비교한다. 이는 경량 암호 알고리즘의 장점이 퇴색될 수 있지만, 게임을 실행하는 PC 하드웨어 스펙이 암호/복호화에 필요한 메모리나 CPU 사이클에 큰 영향을 받지 않을 정도로 발전했기 때문이다.
- 암호 알고리즘을 표준에 맞게 구현했지만, 해당 구현이 최적이라 할 수 없다. 이에 따른 성능 차이가 발생할 수 있다.

1.3 수정 사항

기존 주제인 'AES, ARIA 와 다른 경량 암호 알고리즘의 비교'는 다음과 같은 이유로 변경되었다.

- AES, ARIA 가 다른 경량 암호 알고리즘에 비해 느리다는 근거를 찾을 수 없다.
- 경량 암호 알고리즘은 리소스(CPU, 메모리)가 상당히 제한된 상황에서 사용된다. 즉, 게임을 실행하는 PC 환경에 초점을 두고 만들지 않았다.

위와 같은 이유로 주제를 '온라인 게임에서의 안전한 통신 구현'으로 변경하였다.

2 설계 상세화 및 변경 내역

일관된 테스트를 위해 작업 및 실행 환경을 다음과 같이 통일하였으며, 이후 작성될 테스트 결과에 자세한 하드웨어 스펙을 기재한다.

- OS : Windows 11 x86-64
- Unity Version : 2022.3.30f1
- IDE : Visual Studio 2022
- Compiler : Microsoft Visual C++ (MSVC)

2.1 블록 암호 알고리즘 및 운영모드 설계

- 블록 암호 알고리즘 : AES, ARIA, HIGHT, SPECK, TWINE
- 블록 암호 운영모드 : CBC, CFB, CTR, ECB, GCM, OFB
- Key Size : 128 bits
- Block Size : 64 or 128 bits

AES는 NIST에서 제시한 표준¹⁾을 따라 구현한다. ARIA, HIGHT는 KISA에서 제시한 표준²⁾³⁾을 따라 구현한다. SPECK은 NSA에서 제시한 표준⁴⁾을 따라 구현한다. TWINE은 NEC에서 제시한 표준⁵⁾을 따라 구현한다. 운영모드는 구체적인 블록 암호 클래스에 의존하지 않도록 아래와 같이 IEncryptionAlgorithm 인터페이스에 의존하는 구조로 설계하였다.

// 해당 인터페이스를 블록 암호 클래스가 구현함.

```
public interface IEncryptionAlgorithm
{
    byte[] Encrypt(byte[] plainText);
    byte[] Decrypt(byte[] cipherText);
    string AlgorithmName { get; }
    int GetBlockSize();
}
```

¹⁾ Federal Inf. Process. Stds. (NIST FIPS) - 197 | <https://www.nist.gov/publications/advanced-encryption-standard-aes>

²⁾ 블록암호 ARIA | <https://seed.kisa.or.kr/kisa/Board/19/detailView.do>

³⁾ 블록암호 HIGHT | <https://seed.kisa.or.kr/kisa/Board/18/detailView.do>

⁴⁾ SIMON and SPECK Implementation Guide | <https://nsacyber.github.io/simon-speck/implementations>

⁵⁾ Ultra-lightweight block cipher Twine | <https://www.nec.com/en/global/rd/tg/code/symenc/twine.html>

// 아래 추상 클래스를 블록 암호 운영모드 클래스가 상속받음.

```
public abstract class OperationMode(IEncryptionAlgorithm encryptionAlgorithm) : IOperationMode
{
    public abstract byte[] Encrypt(byte[] plainText);
    public abstract byte[] Decrypt(byte[] cipherText);
    protected byte[] Padding(byte[] input)
    {
        int blockSize = encryptionAlgorithm.GetBlockSize();
        int paddingSize = blockSize - (input.Length % blockSize);
        byte[] output = new byte[input.Length + paddingSize];
        Array.Copy(input, output, input.Length);
        for (int i = input.Length; i < output.Length; i++)
        {
            output[i] = (byte)paddingSize;
        }

        return output;
    }
    protected byte[] UnPadding(byte[] input)
    {
        int paddingSize = input[input.Length - 1];
        for (int i = input.Length - paddingSize; i < input.Length; i++)
        {
            if (input[i] != paddingSize)
                throw new ArgumentException("Invalid padding.");
        }
        byte[] output = new byte[input.Length - paddingSize];
        Array.Copy(input, output, output.Length);

        return output;
    }
    public abstract string ModeName { get; }
    public abstract string AlgorithmName { get; }

    protected readonly IEncryptionAlgorithm encryptionAlgorithm = encryptionAlgorithm;
}
```

각 블록 암호의 패딩 기법은 PKCS#7⁶⁾을 바탕으로 구현하였다.

⁶⁾ PKCS #7: Cryptographic Message Syntax Version 1.5 | <https://datatracker.ietf.org/doc/html/rfc2315>

2.2 공개키 암호 알고리즘 및 서명 설계

공개키 암호 알고리즘 : ECC

공개키 서명 알고리즘 : ECDSA

세션키 교환 알고리즘 : ECDH

사용 해시 함수 : SHA - 256

사용 타원 곡선 : secp256r1 (P - 256)

ECDSA의 경우 NIST에서 제시한 DSS(Digital Signature Standard)⁷⁾를 참고하여 구현하며, 구현 후 테스트는 NIST에서 제시한 표준 테스트 벡터를 사용한다. ECDH의 경우 RFC 8418⁸⁾을 참고하여 구현하며, 구현 후 테스트는 NIST에서 제시한 표준 테스트 벡터⁹⁾를 사용한다. 해시함수와 난수생성은 직접 구현하지 않고 .NET에서 제공하는 SHA256, RandomNumberGenerator 클래스를 사용한다.

ECDSA에 사용할 서명은 X.509를 사용하는 것이 정석이지만, 본 과제에서는 인증 서버의 구현이 큰 의미를 갖지 못하고, 최초 세션 수립 시 서버임을 인증하는 역할만 하면 되기에 해당 내용을 제외하였다.

타원 곡선의 경우 Group을 형성하는 데에 필요한 연산 (Add, Mul, Inverse 등)을 모두 구현할 필요가 있다. 해당 구현에 필요한 큰 수 계산의 경우 .NET에서 제공하는 기본 라이브러리인 BigInteger를 사용하며, 점을 나타내는 구조체인 ECPoint는 다음과 같이 정의한다.

```
public struct ECPoint(BigInteger x, BigInteger y)
{
    public BigInteger X { get; } = x;
    public BigInteger Y { get; } = y;

    public readonly bool IsInfinity => X == 0 && Y == 0;

    public static readonly ECPoint Infinity = new(0, 0);
}
```

여기서 Infinity는 무한원점을 의미한다. 이를 통한 ECPoint간의 연산들을 구현한다.

⁷⁾ Federal Inf. Process. Stds. (NIST FIPS) - 186-4 DSS | <https://csrc.nist.gov/pubs/fips/186-4/final>

⁸⁾ IETF RFC 8418 | <https://datatracker.ietf.org/doc/html/rfc8418>

⁹⁾ NIST CAVP ECC CDHVS | <https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/component-testing>

2.3 통신 프로토콜 설계

서버/클라이언트간 안전한 통신을 위해서 최초 세션 수립 시 세션키를 교환하는 과정이 필요하다. 이는 TLS 1.3¹⁰⁾을 참고하여 구현하였으나, 표준보다 동작을 간소화 하였으며, 특히 Cipher suite을 교환하는 부분을 제거하였다. 아래 그림은 최초 TCP 연결 수립 후 서버 인증과 세션키를 교환하는 과정(handshake)을 담았다.

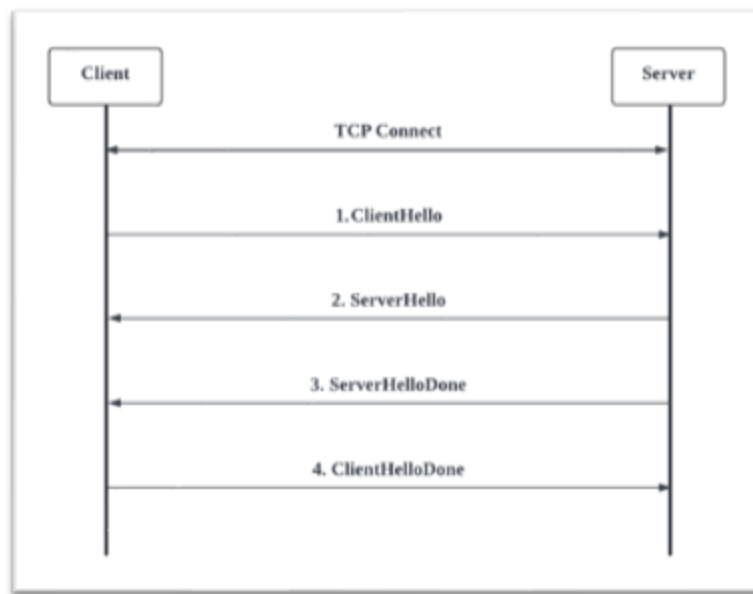


그림 1 - 인증/세션키 교환 과정 (handshake)

1. ClientHello
클라이언트가 공개키/비밀키 쌍을 생성하고, 서버에 공개키를 넘긴다.
2. ServerHello
서버가 공개키/비밀키 쌍을 생성하고, 클라이언트의 공개키를 받아 공유 세션키를 생성한다. 이후 인증서와 공개키를 클라이언트에 넘긴다.
3. ServerHelloDone
서버가 세션키를 통해 상호 합의된 메시지를 암호화 한 후 보내며 handshake 과정을 종료한다. 이 단계는 추후 구현하며 2번 단계와 통합될 수 있다.
4. ClientHelloDone
클라이언트가 서버의 공개키를 받아 공유 세션키를 생성한다. 이후 인증 서버를 통해 인증서를 검증하며 ServerHelloDone에서 받은 메시지를 복호화하여 세션키의 유효성을 검증한다. 정상적으로 복호화가 진행되었을 경우 ClientHelloDone을 송신하며 handshake 과정을 종료한다.

¹⁰⁾ IETF RFC 8446 | <https://datatracker.ietf.org/doc/html/rfc8446>

위 handshake 과정을 통해 세션키를 생성하고, 해당 세션키를 사용해 패킷 암호/복호화를 수행한다. 패킷 구조는 다음과 같이 정의된다.

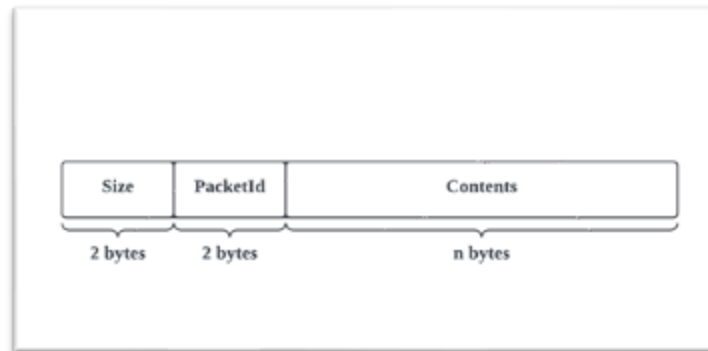


그림 2 - 패킷 구조

최초 4 bytes 는 Size 와 PacketId 가 고정이며, PacketId 는 해당 패킷이 어떤 이벤트인지 알리는 역할을 한다. 모든 패킷은 IPacket 인터페이스를 구현하며, 해당 인터페이스는 다음과 같다.

```
public interface IPacket
{
    ushort Protocol { get; }
    void Read(ArraySegment<byte> segment);
    ArraySegment<byte> Write();
}
```

위 인터페이스를 통해 패킷의 Read/Write 에 필요한 일관된 인터페이스를 구현한다. 패킷의 Contents 부분을 암호/복호화 하며 각 블록 암호 알고리즘과 운영모드별 성능 측정을 시행한다.

2.4 게임 서버 설계

게임 서버는 TCP 소켓 서버이다. 기본적으로 게임 서버가 해야할 일은 기존 TCP 소켓 서버와 크게 다르지 않다. 정상적으로 소켓의 연결을 받고(listen, accept), 클라이언트 세션과의 send/rcv가 정상적으로 이루어지면 된다. 단, 게임 서버인 만큼 성능을 위해 멀티스레드가 필수적이며, 기존의 블로킹 함수들을 사용할 수 없다. 따라서, 비동기 논블로킹 서버를 설계해야 한다.

서버의 I/O 모델은 IO Completion Port 를 채택하였다. 모든 입출력 완료 통지를 비동기적으로 받을 수 있게 되며, Event 기반 모델에 비해 많은 장점이 있다. 먼저, 기존 Event 기반 모델은 입출력 버퍼를 Send/Recv 호출 시 따로 제공할 수 없고, 커널에서 입출력이 끝난 뒤 시스템 버퍼에 있는 내용을 복사해오는 방식이다. 하지만, IOCP 방식은 프로그래머가 전달한 입출력 버퍼에 내용이 바로 쓰여지기에 성능이 더 뛰어나다. 또한 .NET 에서 제공하는 Socket 라이브러리의 Async 함수는 Pending 에 필요한 스레드를 라이브러리에서 운영 체제를 활용해 관리하기에 편리하며 성능이 뛰어나다.

구체적으로 게임 서버가 해야할 일은 다음 그림과 같다.

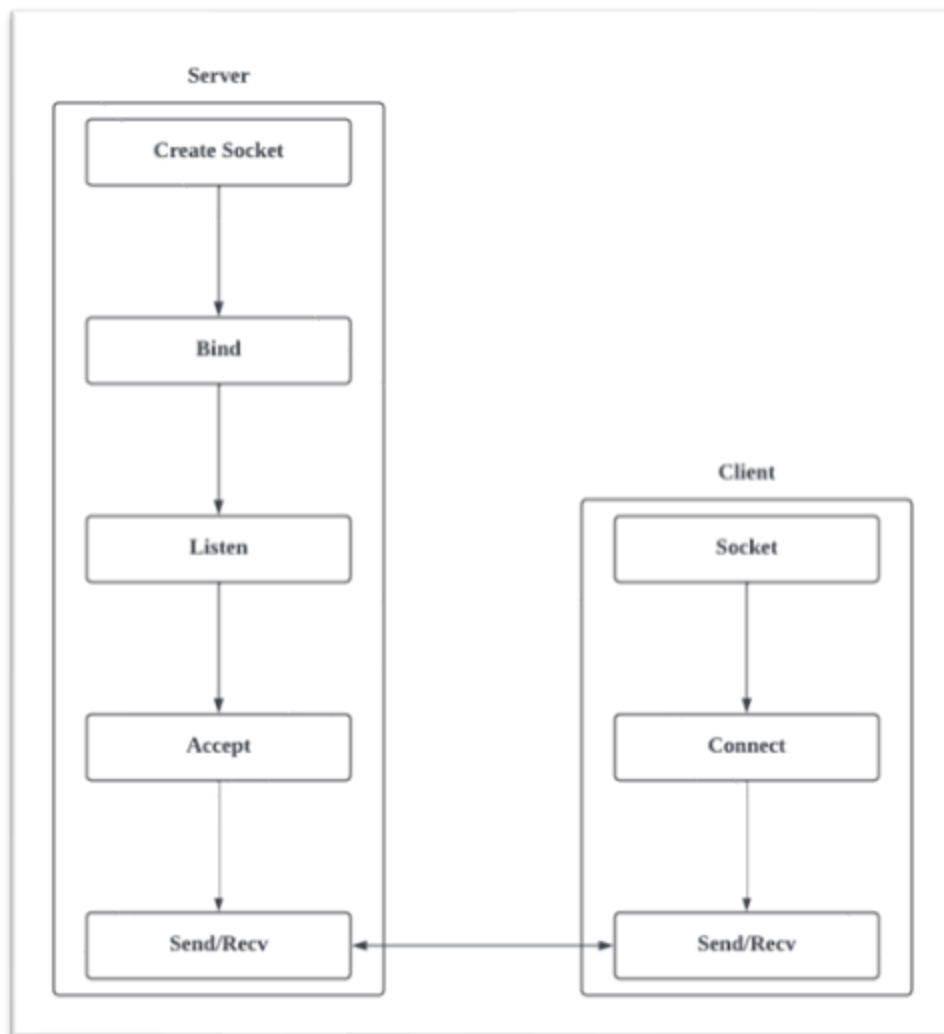


그림 3 - 서버/클라이언트 연결 과정

위 그림에서 볼 수 있듯이, 기본적인 소켓 연결 과정 유사하다. 하지만, 게임 서버는 Accept 함수를 호출하는 동안 해당 스레드가 Blocking 될 경우 성능이 매우 떨어지기에, AcceptAsync 함수를 사용해 연결을 논블록킹 상태로 비동기적으로 처리할 필요가 있다. Send/Recv 또한 마찬가지로 SendAsync/RecvAsync 함수를 사용할 필요가 있다. 이후 클라이언트들이 연결된다면 다음 그림과 같아진다.

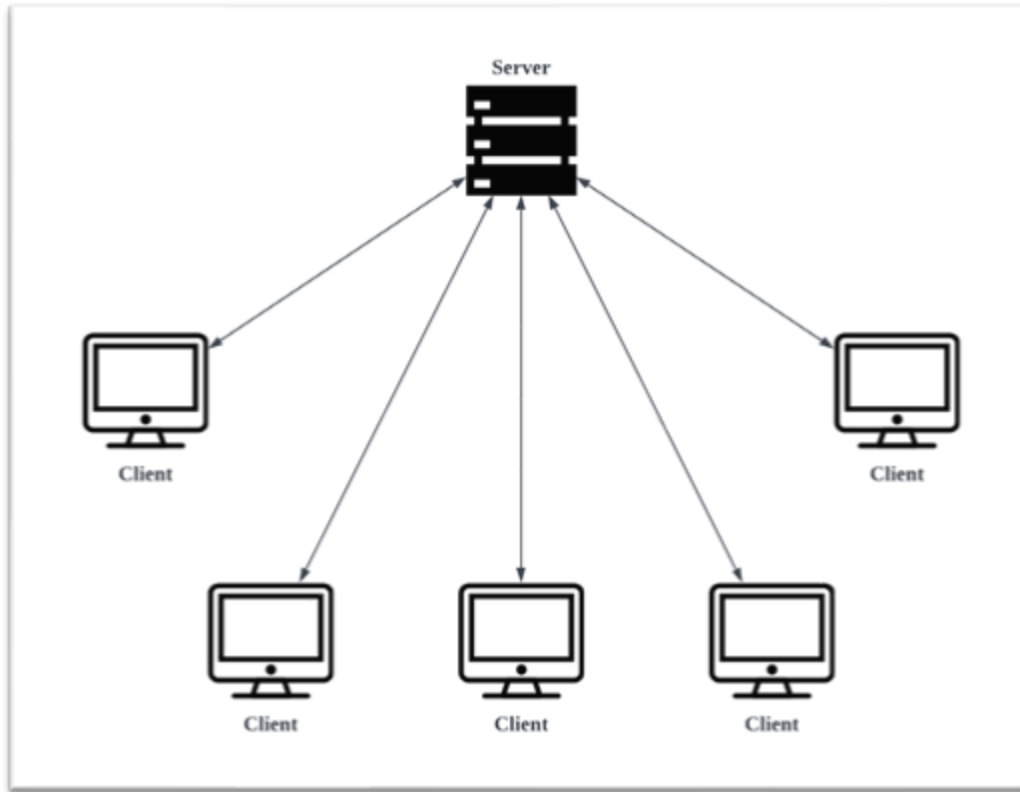


그림 4 - 서버/클라이언트 구조

위 그림에서 볼 수 있듯이, 클라이언트가 다른 클라이언트와 연결되지 않으며, 서버와 클라이언트만 연결된다. 또한, 각 클라이언트가 보내는 패킷을 처리하고, 패킷의 브로드캐스팅/멀티캐스팅이 가능해진다. 본 과제는 여기서 Send/Recv를 하기 전, 헤더를 제외한 패킷의 암호화 시간을 측정하게 된다.

게임의 종류는 여러가지가 있지만, 본 과제는 2D 게임을 바탕으로 다른 플레이어의 위치 정보를 암호화할 것이다. 따라서, 위치 정보 동기화 패킷이 지속적으로 암호화 → Send → Recv → 복호화 구조를 거쳐야 한다.

2.5 게임 클라이언트 설계

게임 클라이언트는 현재 가장 높은 시장 점유율을 가진 Unity 게임 엔진을 선택하였다. 다른 게임 엔진에 비해 가볍고 대중적이며 사용이 편리한 장점이 있다. 대부분의 코드는 서버에서 사용한 코드를 그대로 사용할 수 있지만, 몇 가지 고쳐야 할 부분이 있다.

Unity 에서의 소켓 통신은 일반적인 콘솔 애플리케이션에서의 소켓 통신과 다르다. 콘솔 애플리케이션의 경우 Entry Point 가 Main 함수이지만, Unity 는 MonoBehaviour 라는 Unity 만의 특수한 base class 를 가진다. 이를 상속받는 클래스가 Game Object 에 추가되면 작성된 C# 스크립트가 실행되는 방식이다. 또한, MonoBehaviour 를 상속받는 클래스가 override 할 수 있는 주요 함수 2 가지가 있다.

```
public class Class1 : MonoBehaviour
{
    // 최초 게임 실행 시 실행되는 함수
    void Start()
    {

    }

    // 매 프레임마다 실행되는 함수
    void Update()
    {

    }
}
```

위 코드에서 볼 수 있듯, Start 와 Update 에 소켓 연결에 대한 적절한 코드를 작성하고, 이를 Game Object 에 추가하여 실행하는 것이 Unity 에서 온라인 게임을 설계하는 정석이다. 또한, Update 함수는 매 프레임마다 호출되기에, PC 별 성능에 따라 해당 함수 호출 빈도가 달라진다. 따라서 위치 정보를 보내는 함수를 Update에 작성하는 것은 적절치 않다. 대신, 아래와 같이 Coroutine을 사용한다.

```
public class MyPlayer : Player
{
    NetworkManager _networkManager;

    void Start()
    {
        StartCoroutine("CoSendPacket");
        // 앞서 언급한 소켓 연결 클래스
        _networkManager = GameObject.Find("NetworkManager").GetComponent<NetworkManager>();
    }

    IEnumerator CoSendPacket()
    {
        while (true)
        {
            yield return new WaitForSeconds(0.25f);

            C_Move movePacket = new();
            movePacket.posX = Random.Range(-50, 50);
        }
    }
}
```

```
        movePacket.posY = 0;
        movePacket.posZ = Random.Range(-50, 50);
        // 패킷을 형식에 맞게 직렬화한 후 Send
        _networkManager.Send(movePacket.Write());
    }
}
```

본 과제에서 구현할 게임은 클라이언트 자신의 플레이어 외에 다른 클라이언트의 플레이어의 위치 정보를 동기화 할 필요가 있다. 이는 다음과 같은 방식으로 처리한다.

```
public void Move(S_BroadcastMove packet)
{
    if (_myPlayer.PlayerId == packet.playerId)
    {
        _myPlayer.transform.position = new Vector3(packet.posX, packet.posY, packet.posZ);
    }
    else
    {
        Player player = null;
        if (_players.TryGetValue(packet.playerId, out player))
        {
            player.transform.position = new Vector3(packet.posX, packet.posY, packet.posZ);
        }
    }
}
```

위는 클라이언트가 직접 조작할 수 있는 플레이어와 다른 클라이언트의 플레이어에 붙는 C# 스크립트가 달라지기에 필요한 과정이다.

이후 서버와 마찬가지로 위치 정보 동기화 패킷이 암호화 -> Send -> Recv -> 복호화 구조를 거치며, 여기서 시간 측정을 할 것이다.

3 구성원별 진척도

이름	진척도
고세화	<ul style="list-style-type: none"> ● 대칭키 암호(AES, ARIA, HIGHT, TWINE, SPECK) 구현 완료 ● 공개키 암호(ECC) 구현 완료 ● C# 비동기 게임 서버 설계 및 구현 완료 ● Unity 게임 클라이언트 설계 완료. 구현 진행중 ● 통신 프로토콜 설계 완료. 구현 진행중
김재민	유니티 3D 엔진을 포함한 게임 엔진조사 및 환경 구축. HIGHT 알고리즘 조사 및 패킷 암호화와 관련된 프로그래밍 언어로 구현하고 이를 이용해 유니티 네트워크 환경 구현 보조 작업을 수행
이강인	실시간 상호작용이 많은 유니티에서 경량화 암호 알고리즘을 통한 보안 강화 필요성 조사. AES, ARIA 알고리즘 C# 구현, 경량 암호 알고리즘간 특징 및 성능 비교 수행

4 보고 시점까지의 과제 수행 내용 및 중간 결과

4.1 블록 암호 알고리즘 및 운영모드 구현

블록 암호 알고리즘 AES, ARIA, HIGHT, TWINE, SPECK 과 운영 모드인 ECB, CBC, CFB, OFB, CTR, GCM 을 구현하였다. 각각의 테스트 벡터는 앞서 언급한 것처럼 표준 테스트 벡터이다. 테스트 벡터를 통과할 경우 구현한 블록 암호 알고리즘이 정상적으로 구현되었음을 판단할 수 있다. 기본적으로 블록의 크기가 128 bits 이지만, HIGHT 와 TWINE 의 경우 64 bits 이다. 이에 따라 GCM 모드의 구현이 따로 되어야 하지만, 이는 추후 성능 비교 및 측정에 필요할 때 구현할 것이다.

ECB, CBC, CFB, OFB, CTR 테스트 통과 판별 조건문은 다음과 같다. (AES128-ECB 의 테스트 코드 중 일부이다.) 아래 코드는 CipherText 가 테스트 벡터의 CipherText 와 같은지 비교하며, CipherText 를 Decrypt 하였을 때 PlainText 가 테스트 벡터의 PlainText 와 같은지 비교한다.

```
if (!SequenceEqual(cipherText, vector.CipherText, 16, vector.CipherText.Length + 16)
    || !SequenceEqual(ecb.Decrypt(cipherText), vector.PlainText, 0, vector.PlainText.Length))
{
    // 테스트 실패시 조건문 실행
}
```

GCM 테스트는 앞서 본 5 가지의 운영 모드보다 좀 더 까다로운 조건문을 가진다. 이는 GCM 이 암호화 뿐만 아니라 인증과 무결성의 성질도 같이 갖기 때문이다. 통과 판별 조건문은 다음과 같다.

```
static bool ValidationCheck(AESTestVector vector, bool validation, byte[] plainText, byte[]
cipherText)
{
    // 정상 태그로 판정하고 테스트 벡터의 결과가 비정상 태그인 경우
    if (validation && vector.IsFail)
        return false;

    // 비정상 태그로 판정하고 테스트 벡터의 결과가 정상 태그인 경우
    if (!validation && !vector.IsFail)
        return false;

    // 정상 태그로 판정한 상태에서 암호문과 테스트 벡터의 암호문이 다른 경우
    if (validation && !cipherText.SequenceEqual(vector.CipherText))
        return false;

    // 정상 태그로 판정한 상태에서 복호화한 암호문과 테스트 벡터의 평문이 다른 경우
    if (validation && !plainText.SequenceEqual(vector.PlainText))
        return false;

    return true;
}
```

아래는 테스트 프로그램 실행 화면의 일부이다.

```
[AES - CFB] [Success] CFB128GFSbox128.rsp
[AES - CFB] [Success] CFB128KeySbox128.rsp
[AES - CFB] [Success] CFB128MMT128.rsp
[AES - CFB] [Success] CFB128VarKey128.rsp
[AES - CFB] [Success] CFB128VarTxt128.rsp

[AES - OFB] [Success] OFBGFSbox128.rsp
[AES - OFB] [Success] OFBKeySbox128.rsp
[AES - OFB] [Success] OFBMMT128.rsp
[AES - OFB] [Success] OFBVarKey128.rsp
[AES - OFB] [Success] OFBVarTxt128.rsp

[AES - CTR] [Fail] There is no test vectors.

[AES - GCM] [Success] gcmDecrypt128.rsp
[AES - GCM] [Success] gcmEncryptExtIV128.rsp

***** AES TEST END *****

***** ARIA TEST START *****

[ARIA - ECB] [Success] ARIA128(ECB)KAT.txt
[ARIA - ECB] [Success] ARIA128(ECB)MMT.txt
```

그림 5 - 블록 암호 테스트 벡터 프로그램 실행 화면

기본적으로 통과하지만, 표준에 테스트 벡터가 없었을 경우 테스트할 수 없었다. 다음 표는 각 블록 암호와 운영 모드별 테스트 통과 여부를 나타낸다.

	AES	ARIA	HIGHT	SPECK	TWINE
ECB	O	O	O	O	O
CBC	O	O	O	O	X
CFB	O	O	O	X	X
OFB	O	O	O	X	X
CTR	X	O	O	O	X
GCM	O	O	X	X	X

(O : 통과 X : 테스트 벡터가 없는 상태)

4.2 공개키 암호 알고리즘 및 서명 구현

공개키 암호 알고리즘인 ECC 와 이를 활용한 서명 알고리즘 ECDSA 를 구현하였다. 표준 테스트 벡터를 통해 테스트 하였으며, 테스트 종류는 다음과 같다.

1) KeyPairTest

비밀키에 대한 공개키 생성 테스트이다. 이는 다음과 같은 통과 판별 조건문을 가진다.

```
ECPoint pubKey = ecc.GetPublicKey(vector.Key);
if (pubKey.X != vector.Qx || pubKey.Y != vector.Qy)
    // 테스트 실패 시 조건문 실행
```

비밀키에 대한 공개키 쌍이 테스트 벡터와 다를 경우 테스트는 실패한다.

2) PKVTest

공개키가 유효한지 판단한다. 이는 다음과 같은 통과 판별 조건문을 가진다.

```
int res = ecc.IsValidPoint(new ECPoint(vector.Qx, vector.Qy));
if ( (res == 0 && !vector.IsValid) || ((res == 1 || res == 2) && vector.IsValid) )
    // 테스트 실패 시 조건문 실행
```

만약 ECPoint 가 mod n 공간에 속하지 않거나, 타원 곡선 상에 존재하지 않을 경우 테스트는 실패한다.

3) SigVerTest

서명 검증 테스트이다. 주어진 message, 공개키(Qx, Qy)에 대해 서명(R, S)이 유효한지 검증한다. 이는 다음과 같은 통과 판별 조건문을 가진다.

```
bool res = ecc.Verify(vector.Msg, vector.R, vector.S, new ECPoint(vector.Qx,
vector.Qy));
if ( (res && !vector.IsValid) || (!res && vector.IsValid) )
    // 테스트 실패 시 조건문 실행
```

만약 서명 검증의 결과가 테스트 벡터의 결과와 다르면 테스트에 실패한다.

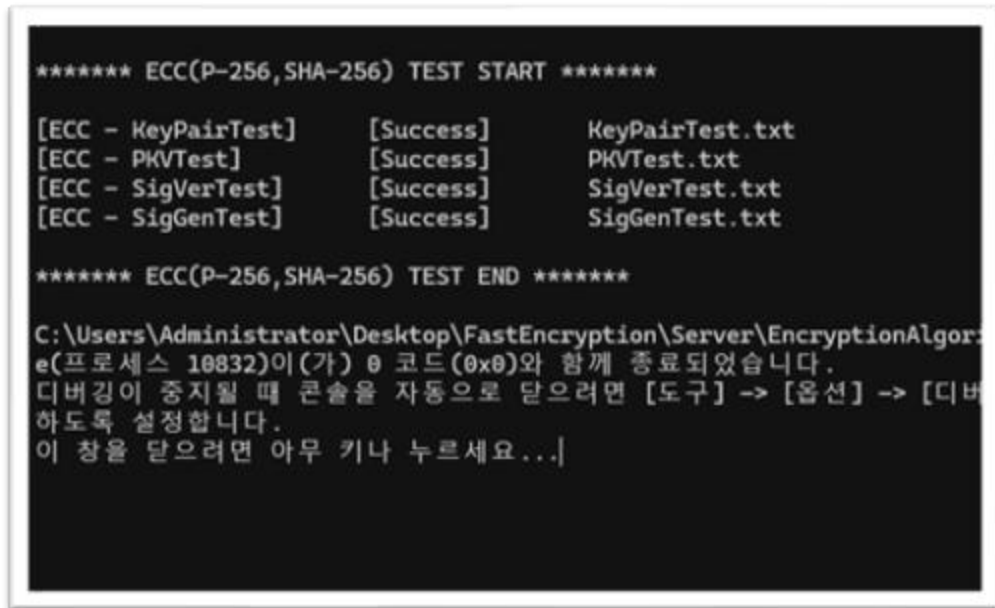
4) SigGenTest

서명 생성 테스트이다. 주어진 message, 비밀키(d), 공개키(Qx, Qy), 난수(k)에 대해 생성한 서명을 테스트한다. 이는 다음과 같은 통과 판별 조건문을 가진다.

```
byte[] signature = ecc.Sign(vector.Msg, vector.Key, vector.K);
if (!signature.SequenceEqual(TestSignature))
    // 테스트 실패 시 조건문 실행
```

만약 서명 생성의 결과가 테스트 벡터에서 주어진 서명(R, S)와 다르면 테스트에 실패한다.

아래는 테스트 프로그램 실행 화면이다.



```
***** ECC(P-256,SHA-256) TEST START *****  
[ECC - KeyPairTest]      [Success]      KeyPairTest.txt  
[ECC - PKVTest]          [Success]      PKVTest.txt  
[ECC - SigVerTest]       [Success]      SigVerTest.txt  
[ECC - SigGenTest]       [Success]      SigGenTest.txt  
  
***** ECC(P-256,SHA-256) TEST END *****  
  
C:\Users\Administrator\Desktop\FastEncryption\Server\EncryptionAlgorithms  
e(프로세스 10832)이(가) 0 코드(0x0)와 함께 종료되었습니다.  
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅]  
하도록 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요...]
```

그림 6 - 공개키 암호 테스트 벡터 프로그램 실행 화면

P-256, SHA-256의 경우에 한하여 NIST에서 제시한 ECDSA에 대한 모든 테스트를 통과하였다.

4.3 게임 서버 구현

클라이언트의 접속을 받고, 패킷의 Send/Recv 가 가능하도록 구현하였다. 이 중 핵심인 Send/Recv 파트 중 Recv 를 확인하겠다.

```
public void Start(Socket socket)
{
    _socket = socket;

    _recvArgs.Completed += new EventHandler<SocketAsyncEventArgs>(OnRecvCompleted);
    _sendArgs.Completed += new EventHandler<SocketAsyncEventArgs>(OnSendCompleted);

    RegisterRecv();
}
```

Overlapped I/O나 IOCP 에서 비동기로 입출력을 처리할 경우, 해당 비동기 입출력이 끝났을 때 작업을 처리해 줄 콜백 함수가 반드시 있어야 한다. 위 코드에서 OnRecvCompleted와 OnSendCompleted가 앞서 말한 콜백 함수이다.

```
void RegisterRecv()
{
    if (_disconnect == 1)
        return;

    _recvBuffer.Clean();
    ArraySegment<byte> segment = _recvBuffer.WriteSegment;
    _recvArgs.SetBuffer(segment.Array, segment.Offset, segment.Count);

    try
    {
        bool pending = _socket.ReceiveAsync(_recvArgs);
        if (pending == false)
        {
            OnRecvCompleted(null, _recvArgs);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine($"RegisterRecv Failed {e}");
    }
}
```

RegisterRecv에서 Recv에 사용할 버퍼를 설정한 뒤, ReceiveAsync를 통해 클라이언트에게서 패킷을 받을 때 까지 Pending 하게 된다. 이후, 클라이언트에서 패킷을 보내면 앞서 설정한 콜백 함수가 호출된다. 아래는 콜백 함수의 코드이다.

```
void OnRecvCompleted(object sender, SocketAsyncEventArgs args)
{
    if (args.BytesTransferred == 0)
    {
        // 연결 종료
        Console.WriteLine("args.BytesTransferred == 0, Disconnect");
    }
}
```

```

        Disconnect();
        return;
    }

    if (args.SocketError != SocketError.Success)
    {
        // 소켓 에러 발생
        Console.WriteLine("args.SocketError != SocketError.Success, Disconnect");
        Disconnect();
        return;
    }

    try
    {
        if (_recvBuffer.OnWrite(args.BytesTransferred) == false)
        {
            // 버퍼 여유 공간 부족
            Disconnect();
            return;
        }

        int processLen = OnRecv(_recvBuffer.ReadSegment);
        if (processLen < 0 || _recvBuffer.DataSize < processLen)
        {
            // 처리한 데이터가 실제 버퍼에 있던 데이터보다 많을 경우
            Disconnect();
            return;
        }

        if (_recvBuffer.OnRead(processLen) == false)
        {
            // 읽은 데이터가 실제 버퍼에 있던 데이터보다 많을 경우
            Disconnect();
            return;
        }

        RegisterRecv();
    }
    catch (Exception e)
    {
        Console.WriteLine($"OnRecvCompleted Failed {e}");
    }
}

```

위 코드에서 에러 없이 정상적으로 받은 패킷이 처리되었을 경우, 다시 RegisterRecv를 호출하며, ReceiveAsync를 통해 Pending 상태가 된다.

비동기 입출력 함수들의 경우 대부분 위와 같은 구조를 따르며, RecvAsync 외에도 SendAsync, AcceptAsync, ConnectAsync 를 사용하여 기존 블로킹/동기 방식의 소켓 서버에 비해 상당한 수준의 성능을 얻을 수 있었다.

4.4 게임 클라이언트 구현

게임 클라이언트의 서버 접속과 위치 정보 동기화 패킷 핸들링을 구현하였다. 네트워크 관련 코드들은 서버에서 사용한 것을 어느정도 재사용 할 수 있었다. 단, Unity API 는 메인 스레드에서만 호출 가능하기에 플레이어 위치 정보 동기화 패킷 핸들링에서 수정이 필요하다.

아래는 11 개의 클라이언트가 각자의 위치 정보 패킷을 서버에 보내고, 서버가 이를 다른 클라이언트에 브로드캐스팅하는 모습이다. 위치 정보 패킷은 초당 4 개씩 보낸다.



그림 7 - 클라이언트 실행 화면

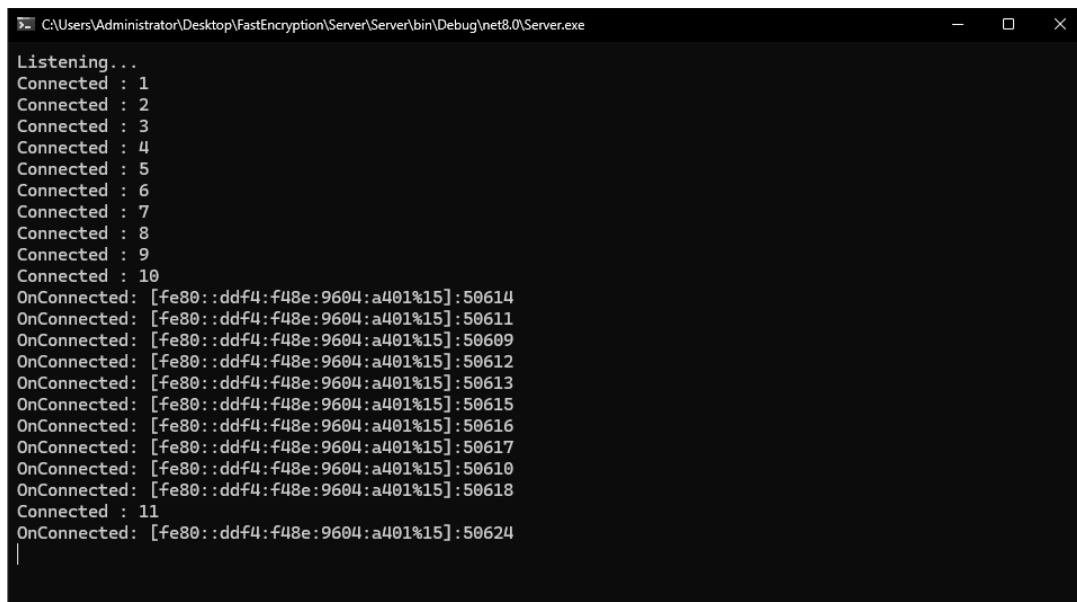


그림 8 - 서버 실행 화면

5 멘토의견서 피드백 반영사항

- 1) 기존의 주장인 '다른 경량 암호 대비 AES 암호화로 인하여 통신에 지연이 발생한다'는 근거가 부족하기에 주제를 변경하였다.
- 2) 통신 프로토콜이 구체적으로 설계되지 않았기에, 최초 세션 수립 시 세션키 교환부터 설계하였다.
- 3) 암호 알고리즘 성능 비교를 위해 암호화 - 통신 - 복호화에 걸리는 시간을 모두 측정해야 한다. 이를 와이어 샤크를 통해 측정할 경우 암호 알고리즘에 따른 속도 변화를 측정하기 어렵다. 따라서, 서버와 클라이언트는 각 패킷을 암호/복호화 할때의 시간을 측정하며, 기존 지연 시간 대비 추가되는 지연 시간을 계산할 것이다.
- 4) 경량 암호 알고리즘은 리소스가 제한된 상황에서 사용하도록 개발되었으며, 현재 주장하는 'PC 통신에 사용할 경우 AES 대비 가볍고 빠르다' 라는 주장에 근거가 부족하다. 단, 게임 통신에 경량 암호 알고리즘을 사용한 사례를 찾아보기 어렵기에 본 과제에서 수행해볼 것이다.