# DAY4 과제 2번

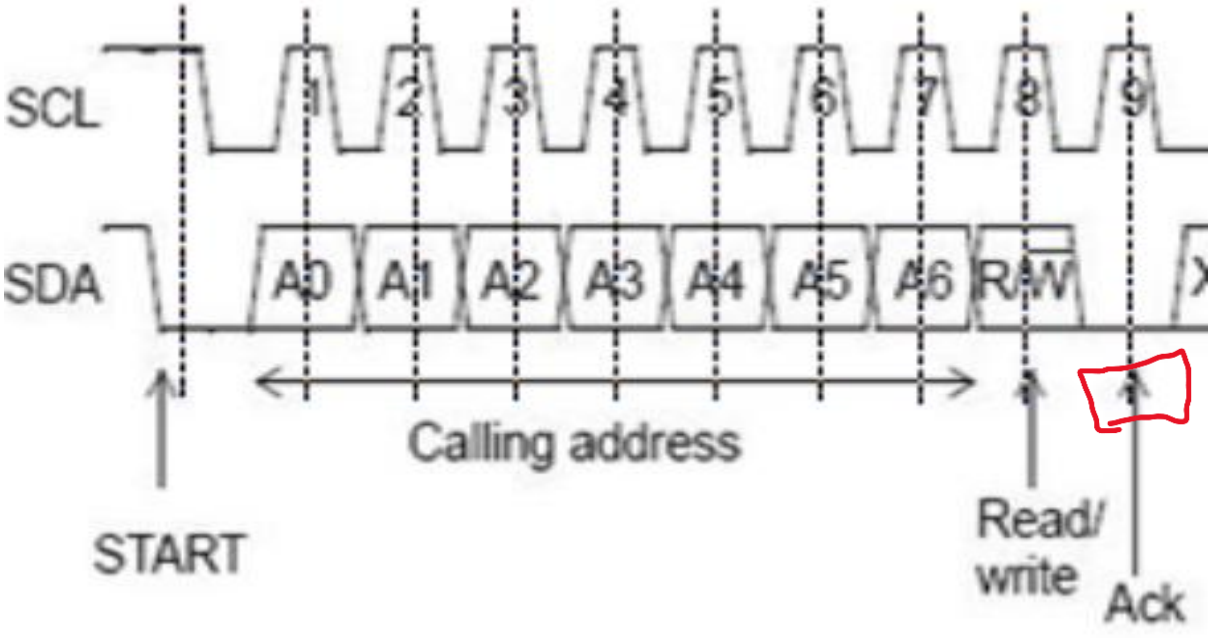
2023741024
로봇학부 박건후

목차

(1) UDP 개요 및 역사

(1-1) UDP 개요

UDP는 비연결형, 신뢰성이 없는 전송 프로토콜입니다. UDP는 User Datagram Protocal의 약자로, 인터넷 프로토콜 스위트에서 전송 계층(Transport Layer)에 위치하는, 비연결형, 비상태, 메시지 지향(datagram-oriented) 전송 프로토콜이에요. 즉 연결되어 있든 아니든 전송을 할 수 있고 그에 따라 수신이 잘 되는지 확인할 수 없어 신뢰성이 없습니다. 그런 특성을 가지는 통신 규약, 통신 규칙이 UDP라고 불리는 것입니다. 이와 반대 예시로 I2C를 예시




로 들 수 있을 것 같습니다.

I2C는 데이터를 받는 쪽이 데이터를 1byte 받고 받았다는 신호인 Ack 신호를 보냅니다. 이처럼 받았다면 받았다고 신호를 데이터를 보내는 쪽에 보냄으로써 통신이 잘 되고 있다는 것을 매번 확

인하며 안정적이고 신뢰성 높은 통신을 할 수 있습니다.

반면, UDP는 Ack처럼 수신자로부터 통신을 받는 것을 하지 않고 오로지 자신의 데이터만 보냅니다. 그렇게 되면 최소한의 오버헤드로 데이터를 빠르게 전송할 수 있게 됩니다. 오버헤드란 소요되는 시간이나 메모리 등을 말하는데 Ack처럼 수신자로부터 통신을 받는 것 또한 시간, 공간, 코드 실행 상 오버헤드가 걸릴 수밖에 없습니다. 하지만 UDP는 수신자로부터 받는 통신을 없애 위와 같은 오버헤드를 없애고 데이터를 빠르고 간단하게 전송할 수 있게 된 것입니다. 이처럼 UDP의 목표는 단순합니다. 최소 오버헤드로 애플리케이션이 만든 메시지(데이터그램)를 빠르게 목적지에 전달하는 것입니다. 이 외에도 UDP는 패킷 기반으로 작동하며 오류 검출을 checksum으로만 수행합니다.

(1-2) UDP 역사

UDP는 1980년에 UDP 표준 문서 RFC 768 발표되고 오늘날까지 이어지는 8바이트 헤더·의사 헤더 기반 체크섬 규칙이 이때 확립되었습니다. 초창기 TCP는 지금의 IP 기능까지 품던 시절이 있었고, 계층 분리 원칙이 강조되면서 IP(네트워크 계층)와 TCP(전송 계층)로 분화. 이때 "가벼운 전송" 수요를 충족하려 UDP 가 함께 자리 잡았습니다. 그 후에 RFC 768이 승인되어 UDP의 정의·헤더 형식·체크섬 규칙이 확정되었습니다. 오늘 기준으로도 놀라울 만큼 한 장 반 분량의 간결한 사양을 갖고 있습니다.

그 후에 UDP가 많이 사용되고 대중화되게 된 경우가 있었습니다.

DNS: "짧고 잦은 요청/응답"이라는 **트랜잭션 지향** 특성에 UDP

가 딱 맞았습니다. 다만, 응답이 크거나 영역 전송(zone transfer) 등은 TCP 사용했습니다.

RTP(2003): 오디오/비디오 같은 실시간 스트림은 "약간의 손실은 괜찮지만, 지연되면 안 되기 UDP 위에서 동작하는 모델이 사실상 표준이 되었었습니다.

그 이후, UDP의 단점인 데이터의 불신뢰성을 사용하려는 시도가 있었습니다.

무선·손상 허용 네트워크 대응 (2004): UDP-Lite: 비트 오류가 일부 있어도 프레임 전체 폐기 대신 '부분 유효' 수용이 더 이득인 환경(예: 음성/영상 코덱의 강건성 활용)에 맞춘 파생 프로토콜.

그 후 현재는

현재: QUIC/HTTP3: TCP 커널 스택의 한계를 넘고, 손실 복구·혼잡 제어·보안(TLS 1.3)·0-RTT 재개 등을 사용자 공간에서 빠르게 진화시키기 위해 UDP를 '운반체'로 선택. 오늘날 브라우저/클라우드가 본격 채택.

까지 오며 UDP의 최대 장점인 속도를 이용하려는 시도가 계속 진행되었음을 알 수 있습니다.

이처럼 UDP는 단순성으로부터 오는 속도를 최대한 활용하여 낮은 지연, 낮은 오버헤드라는 강점을 40년 동안 유지해왔습니다. 그리고 체크섬 의무화를 규정하여 안정성과 효율성의 균형을 잡는 것 등의 진화를 해왔습니다. 또한, 웹에서는 HTTP/3는 TCP에

서 벗어나 UDP 기반 QUIC 로 이동하여 HOL(blocking) 회피, 커넥션 마이그레이션, 낮은 지연을 실현시킴으로써 웹에서도 UDP 를 사용하여 웹 환경에 변화를 주는데 기여를 했습니다.

(2)OSI 7계층 모델

OSI 7계층이란 국제표준화기구(ISO)가 만든 네트워크 통신을 7개의 논리적 층으로 나눈 개념 모델입니다. 실제 인터넷 스택(TCP/IP)과 1:1로 같지는 않지만, 역할 분리, 문제 진단, 설계 의사소통에 표준처럼 쓰입니다. 이 모델의 목적은 통신 기능을 모듈화해 **역할을 분리하는 것,** 계층 간 **표준 인터페이스** 제공하는 것, 각 계층을 **독립적으로 개선/교체** 가능하다는 것 등이 있습니다. 그리고 각각의 계층의 데이터 처리 방식을 보면, L4는 세그먼트(TCP), 데이터그램(UDP), L3은 패킷, L2는 프레임, L1은 비트로, 해당 통신 프로토콜에 맞게 각각의 단위에 맞게 데이터를 전송하는 것입니다. 각각의 계층에 대해 더 자세히 설명드리겠습니다.

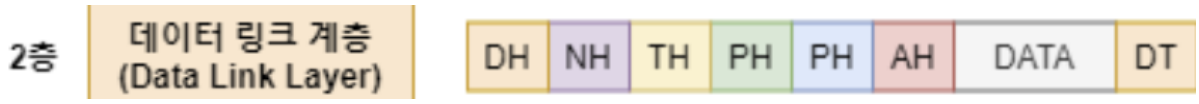

L1: 물리 계층

물리 계층은 0/1비트를 전기, 광, 무선 신호로 실어 보냅니다. 이 때는 대역폭, 비트 오류율, 코딩, 케이블 등이 중요해지고 케이블 불량이나 반사, 감쇠 때문에 통신에 문제가 생기는 경우가 많습니다.


L2: 데이터 링크 계층

같은 링크 내에서 프레임을 전달합니다. 같은 네트워크(같은 스위치/같은 VLAN) 안에서 프레임을 MAC 주소 기준으로 정확히 전달하고, 오류를 검출한다. 구간프레임화를 하고, 오류 검출을 하고 흐름을 제어하는 등의 작업을 합니다. 이렇게 같은 도메인을 사용하는 것 안에서 프레임이라는 데이터 단위를 전달하며, 프레임화, 오류 검출, MAC주소 학습 등을 하는 것입니다. 이를

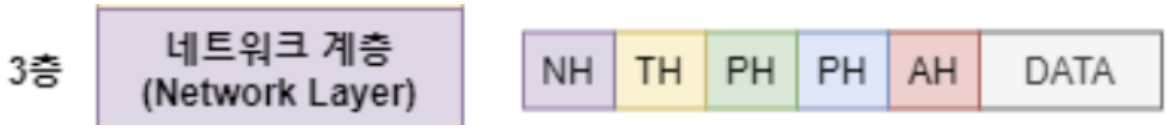
활용하는 기술들이 이더넷, 와이파이 등이 있습니다




L3: 네트워크 계층

서브넷을 넘어 패킷을 IP주소와 라우팅으로 가장 적절한 경로로 전달하는 통신 방식입니다. 여

기서는 데이터 단위를 패킷으로 하여 IP 주소를 사용해 다른 서브넷으로 가는 경로(라우팅) 를 결정하고, 각 라우터는 라우팅 테이블 로 다음 hop 을 정해 패킷을 전달합니다. 이 과정에서 TTL 감소, ICMP 오류 통지, (IPv4에서) 조각화 와 같은 안전장치가 작동합니다. 결과적으로 L3는 서브넷 간, 혹은 인터넷 전역으로 통신을 하거나 Inter-VLAN 통신, NAT를 통한 인터넷 공유, 멀티홉/멀티캐스트/VPN 같은 다양한 통신을 가능하게 합니다.

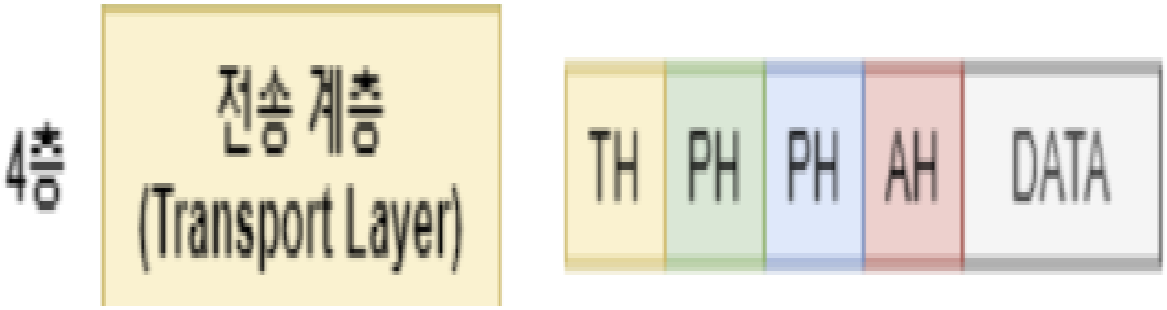



L4: 전송 계층

이번에는 컴퓨티끼리 통신하는 것이 아닌 프로그램끼리 통신하도록 해주는 계층입니다. 이 때 쓰는 주소가 포트, 주고받는 단위가 세그먼트나 데이터 그램입니다. L4가 하는 역할로는 다음과 같습니다.

- 다중화/역다중화: 한 PC에서 여러 앱이 동시에 통신. L4가 포트 번호로 구분해 줍니다.

- 신뢰성/순서/재전송(TCP): 잃어버린 데이터는 다시 보내고, 순서대로 도착하게 만듭니다.
- 흐름 제어(TCP): 받는 쪽 버퍼가 넘치지 않게 속도를 조절합니다.
- 혼잡 제어(TCP): 네트워크가 막히면 속도를 줄이고, 풀리면 늘립니다.
- 오류 검출(UDP/TCP): 체크섬으로 손상 여부를 검출합니다 (복구는 TCP만 자동).
- 메시지 경계 유지(UDP): 보낸 덩어리(데이터그램) 그대로 경계 보존합니다.




그리고 통신의 주소 체계인 포트를 보면, 0~65535(16비트) 사이에서

**Well-known**: 0–1023

**Registered**: 1024–49151

**Ephemeral(임시)**: 49152–65535 (클라이언트가 그때그때 잡아 쓰는 임시 출발 포트)

으로 나뉘며, Well-Known은 이미 기업이나 특정 애플리케이션을 위해 예약한 포트입니다. 일반적으로 권한이 높은 사용자만 사용할 수 있습니다. 그렇기에 사용자가 특정 사이트로 접속하는 것이 아닌 이상 Well-Known으로 정하여 사용자들끼리 통신할 수는 없고, 1023 이상의 값들은 모두 사용 가능하며 사용자들끼리 통신할 때에도 문제없이 작동합니다. 사실 Registered에서도 특정 애플리케이션을 위해 등록된 포트도 몇 있습니다. 예를 들어 포트 8080은 HTTP의 대체 포트로 자주 사용됩니다.

| 프로토콜 종류 | TCP | UDP |
| --- | --- | --- |
| 연결 방식 | 연결형 서비스 | 비연결형 서비스 |
| 패킷 교환 방식 | 가상 회선 방식 | 데이터그램 방식 |
| 전송 순서 | 전송 순서 보장 | 전송 순서가 바뀔 수 있음 |
| 수신 여부 확인 | 수신 여부 확인 | 수신 여부 확인하지 않음 |
| 통신 방식 | 1:1 통신 | 1:1 or 1:N or N:N 통신 |
| 신뢰성 | 높다 | 낮다 |
| 속도 | 느리다 | 빠르다 |

L4인 TCP, UDP를 보겠습니다. TCP는 연결형이고, 신뢰성이 높은 통신 방식으로 손실 시 재전송, 순서 보장, 중복 제거 등의 작업을 할 수 있습니다. 또한 흐름 제어 신호를 보내고, 혼잡한 것을 감지하고 이를 조절하며 신호를 보낼 수 있습니다. 그리고 스트림을 통해 연속적으로 바이트를 보냅니다.

반면, UDP는 비연결형이고, 스트림을 사용하지 않기에 연속적으로 바이트를 보내지 않고 데이터에 경계가 존재합니다. 그리고 신뢰, 순서, 혼잡 제어가 없습니다. 오로지 체크섬으로 오류 검출만 하는 것입니다.

이 UDP를 사용하는 곳은 실시간 스트리밍, VoIP, 온라인 게임 등에서는 지연이 적기에 이를 많이 사용하고, DNS 질의 같은 짧고 잦은 요청/응답이 있는 곳, QUIC(HTTP/3)처럼 UDP 위에 신뢰, 혼잡, 보안을 얹는 현대 프로토콜로 현대적으로는 UDP의 속도를 사용하며 신뢰, 혼잡, 보안 기능을 넣어 안정성을 확보하는 식으로 사용하고 있습니다.

(3) Datagram이란

Datagram이란 한 번에 보내는 경계가 있는 독립적인 메시지 조각입니다. 각 조각은 자기 혼자 목적지까지 갈 수 있을 만큼의

정보(목적지 주소 등)를 가지고 있어서, 중간 장비가 별도 연결 상태 없이 그 조각만 보고 전달할 수 있습니다. 그 외의 핵심 특징으로 정리하면 다음과 같습니다.

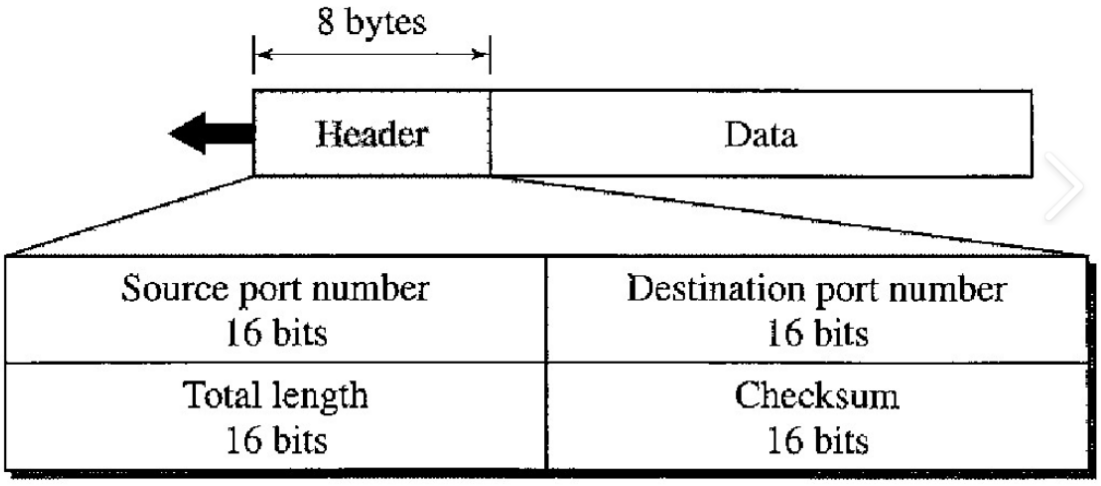
**비연결성**

먼저 연결(handshake)을 만들지 않고도 바로 보낼 수 있습니다. 라우터/스위치는 적힌 주소만 보고 다음으로 넘기게 됩니다.

**독립성**

같은 목적지로 여러 개를 보내도 **순서 보장이 안 되고**, **중간에 몇 개 유실될 수 있습니다**.

*User datagram format*




위 사진은 하나의 데이터 그램을 나타내는 것입니다. 헤더에서 8바이트로 구성이 되고 그 안에 출발 포트, 목적지 포트, UDP + 데이터(payload)의 합이 Total length값이 됩니다. 이 값은 UDP 영역의 총 길이를 정확히 알려줘서, 수신 측이 경계를 잘라내는 데 사용됩니다.

마지막으로 checksum입니다. 이는 오류 검출용 값으로 사용되고, 계산할 때 의사(pseudo) 헤더(출발/목적 IP, 프로토콜 번호, UDP 길이) + UDP 헤더 + 데이터를 1의 보수 합으로 더해 만듭니다.

(4) QT UDP 관련 라이브러리 및 구현

(4-1) QT UDP 관련 라이브러리

- QHostAddress

IP 주소를 표현하는 클래스 ex) QHostAddress 객체 (192.168.188.100);

- QUdpSocket : UDP 소켓 역할

bind(QHostAddress addr, quint16 port, BindMode mode=DefaultForPlatform) : UDP 소켓을 해당 IP, 포트에 연결합니다. 성공 시 true를 반환합니다. 송수신 측 모두에게 포트 고정이 필요할 때 사용합니다. 그리고 BindMode에서는 DefaultForPlatform, ShareAddress(여러 소켓이 같은 포트 공유), DontShareAddres(해당 소켓만 포트 독점) 등등의 모드가 있어서 상황에 따라 원하는 동작을 내도록 할 수 있습니다.

```
text_socket = new QUdpSocket(this);
```

```
if(text_socket->bind(OPERATOR_IP, TEXT_PORT, QUdpSocket::ShareAddress))
{
  connect(text_socket, SIGNAL(readyRead()), this, SLOT(udp_read()));
}
```

위의 사진을 보시면 UdpSocket 객체를 동적할당하고 bind메서드를 호출하여 text_socket의 소켓을 IP, Port와 연결하고 플래그를 ShareAddress로 했음을 알 수 있습니다.

- QBytearray

바이트(8-bit) 배열 컨테이너로써 텍스트든 바이너리든 **그냥 바이트로** 담을 수 있습니다.

**검색/분해**: contains, indexOf, lastIndexOf, startsWith, endsWith, split(char)(→ QByteArrayList)

**대소문자/정리**: toLower(), toUpper(), trimmed(), simplified()

**숫자 변환**: QByteArray::number(123), ba.toInt(&ok, base), toDouble(&ok)

그 외: append: 뒤에 붙이기, prepend: 앞에 붙이기, insert: 위치 삽입, replace:치환, remove: 구간 삭제, resize: 크기 변경, 메모리 확보

위의 것처럼 기본 API가 있고, 그 외에 toBase64()처럼 인코딩 같은 설정도 가능합니다.

- pendingDatagramSize()

다음에 읽을 데이터그램의 길이를 qint64 타입으로 반환합니다. 버퍼를 정확한 크기로 준비하고 싶을 때 사용하며, readDatagram을 호출하기 전에 이를 사용하여 데이터 손실을 없게 해야 합니다.

- bool QUdpSocket::hasPendingDatagrams() const

읽을 데이터그램이 1개 이상 대기 중이면 true, 없으면 false를 반환합니다. 반복 읽기 여부를 판단할 때 if문 조건식에 넣어서 사용합니다.

- readDatagram(char **data*, qint64 *maxSize*, QHostAddress **address* = nullptr, quint16 **port* = nullptr)

maxSize의 바이트값보다 적게 데이터 그램을 받아오고 이를 data 변수안에 저장합니다. 그리고 발신자의 IP 주소를 address라는 포인터 변수에 저장하고, port변수에 발신자 포트를 저장합니다. 만약 **maxSize가 실제 데이터그램 크기보다 작다면** 나머지 데이터는 **버려지게 됩니다.** 이를 방지하려면 **읽기 전에** 반드시 **pendingDatagramSize()(대기 중인 다음 데이터그램의 크기 반환)**를 호출해 버퍼를 미리 준비해야 합니다.

```
QByteArray buffer;
buffer.resize(text_socket->pendingDatagramSize());
text_socket->readDatagram(buffer.data(), buffer.size(), &ROBOT_IP,
&TEXT_PORT);
```

위 코드처럼 buffer를 resize를 하여 크기를 늘려 데이터가 손실되지 않게 하는 것입니다.

- qint64 QUdpSocket::writeDatagram( const char *data, qint64 size, const QHostAddress &address, quint16 port);

이는 char* 포인터 버전으로 버퍼와 길이를 갖고 있을 때, 그리고 보낼 바이트 버퍼와 길이, 목적지 IP/포트를 넣어주면 송신할 수 있습니다.

- qint64 QUdpSocket::writeDatagram(const QByteArray &datagram, const QHostAddress &host, quint16 port);

이는 payload를 QByteArray로 다룰 때이고 바이트 버퍼를 직접 만들 필요가 없다는 것이 장점인 거 같습니다.

```
void MainWindow::udp_write(QByteArray text, uint16_t port,  QUdpSocket &socket)
{
  QByteArray packet;
  packet.push_back(text);
  socket.writeDatagram(packet, ROBOT_IP, port);
}
```

위 사진대로, payload를 packet에 넣고 writeDatagram을 호출하여 text의 값을 송신하는 부분입니다.

**bool joinMulticastGroup(const QHostAddress &groupAddress)**

OS가 고르는 **기본 인터페이스**로 groupAddress 멀티캐스트 그룹에 **가입**.

- **전제(중요)**: 소켓이 **BoundState(이미 bind 완료)** 여야 함.
- **IPv4 주의**: IPv4 그룹에 가입하려면 **QHostAddress::AnyIPv4로 bind** 해야 함.
  (Any로 IPv6 듀얼모드 바인드하면 실패 가능)
- **IPv6 주의**: 인터페이스를 지정하지 않은 **IPv6 멀티캐스트 가입은 OS마다 미지원**일 수 있음 → 인터페이스 지정 오버로드 사용 권장.
- **반환**: 성공 true, 실패 false (+ error() 설정).

- QUdpSocket::receiveDatagram(qint64 maxSize = -1)

receiveDatagram()은 **대기 중인 UDP 데이터그램 1개를 통째로 읽어서** QNetworkDatagram 객체로 돌려줍니다. **데이터(페이로드)** 뿐 아니라 **발신자 주소/포트**, 가능하다면 **수신 시점의 목적지 주소/포트, hop 관련 정보, 수신 인터페이스** 같은 **데이터**도 함께

담기게 됩니다. 이 때 유요하지 않더라도 QNetworkDatagram을 돌려주므로 isValid()로 꼭 반드시 확인해봐야 합니다.

또한 maxSize값으로 동작을 바꿀 수도 있습니다. maxSize값이 -1일 경우 전체 데이터 그램을 읽으려 시도합니다. 그리고 0일 때는 데이터 그램을 버립니다. 일부러 버퍼를 비우고 싶을 때 사용합니다. 그리고 0보다 클 경우 maxSize값 바이트를 읽습니다. 만약 maxSize값보다 더 클 경우에 그 때 나머지는 폐기됩니다. 데이터 손실을 피하려면 maxSize를 직접 제한하는 것이 아닌 pendingDatagramSize()로 크기를 먼저 확인하는 것이 좋습니다.

**QNetworkDatagram 안에 담기는 것들:**

수신 페이로드(QByteArray)

**보낸 쪽** IP/포트

**수신 시점의 목적지(내 쪽) 주소/포트**

**hop 관련 정보**, **수신 당시의 값**을 제공할 수 있음

**어느 네트워크 인터페이스로 받았는지** (멀티 NIC에서 중요)

등의 값을 받아올 수 있습니다.

즉, receiveDatagram()은 **데이터 + 출발지 + 목적지, 인터페이스, hop 정보**까지 한 번에 가져오는 메서드입니다.

| 항목 | readDatagram() | receiveDatagram() |
| --- | --- | --- |
| 버퍼 | **직접** 버퍼를 미리 할당해야 함 | 버퍼를 가진 **객체**를 그대로 반환(기본값이면 전체 읽기 시도) |
| 데이터 | 출발지 주소/포트만 주로 사용 | 출발지 + **목적지/인터페이스/hop** 등 **정보** 보존 |
| 실수 여지<br>(버퍼 용량 부족) | 버퍼 크기 부족 시 **잘림** | maxSize=-1이면 잘림 위험 줄어듦<br>출발지 + **목적지/인터페이스/hop** 등 **정보** 보존 |
| 가독성 | C 스타일 | 객체형(유지 보수 유리) |

readDatagram()과 receiveDatagram()을 표로 비교 및 정리해보았습니다. 이 둘을 비교했을 때 receiveDatagram이 처음에는 다루기 힘들 수도 있으나 안전 장치도 많고 받을 수 있는 정보도 많다는 것을 알게 되었습니다.

(4-2) 구현

```
    // 새로운 UDP 소켓 생성
    udpSocket = new QUdpSocket(this);

    // 지정된 포트로 바인딩 시도
    if (udpSocket->bind(QHostAddress::Any, myPort)) {
        // 데이터 수신 이벤트 연결
        connect(udpSocket, &QUdpSocket::readyRead, this, &MainWindow::readyRead);

        // 연결 상태 업데이트
        isConnected = true;
        ui->statusLabel->setText(QString("연결됨 - 내 포트: %1, 상대방: %2:%3")
                                 .arg(myPort).arg(targetIP).arg(targetPort));

        // 메시지 전송 기능 활성화
        ui->sendButton->setEnabled(true);
        ui->messageInput->setEnabled(true);
        ui->messageInput->setFocus();

        // 연결되고 있는지 메시지 출력
        ui->chatDisplay->append(QString("포트 %1에서 연결 대기 중...").arg(myPort));
    } else {
        // 연결 실패 시 경고 메시지
        QMessageBox::warning(this, "연결 오류", QString("포트 %1에 바인딩 실패").arg(myPort));
    }
}
```

이 부분은 connectToNetWork라는 메서드의 구현부입니다. 여기에서 bind를 하는 부분에서 QHostAddress::Any를 써서 모든 IP4 주소에서 들어오는 패킷을 수신하도록 했습니다. 그래서 해당 포트로 들어오는 패킷을 모두 수신받을 수 있도록 IP를 그렇게 바인딩한 것입니다. Any로 한 이유는 상대방의 들어온 포트를 동적으로 파악하기 위함입니다. 상대방의 IP를 알지 못하는 상태에서 런타임에서 이를 확정짓고 파악하기 위해서는 처음에 바인딩할 때는 모든 IP가 들어올 수 있도록 설정하고 한 번 입력이 들어오면 이를 TargetIP로 저장하고 이 TargetIP를 readyRead 메서드에

서 TargetIP만 입력받도록 설정하여 두 번째 입력부터는 상대방 IP값에 따른 값만 받을 수 있도록 만들었습니다.

```
    // 입력된 메시지와 현재 시간 가져오기
    QString message = ui->messageInput->text().trimmed();
    QString currentTime = QTime::currentTime().toString("hh:mm:ss");
    QByteArray data = message.toUtf8(); // UTF-8로 인코딩

    // UDP 데이터그램 전송
    qint64 result = udpSocket->writeDatagram(data, QHostAddress(targetIP), targetPort);

    if (result != -1) {
        // 전송 성공 시 채팅창에 표시
        ui->chatDisplay->append(QString("[%1] 나: %2").arg(currentTime, message));
        ui->messageInput->clear(); // 입력 필드 비우기

        // 채팅창을 가장 아래로 스크롤
        ui->chatDisplay->verticalScrollBar()->setValue(
            ui->chatDisplay->verticalScrollBar()->maximum());
    } else {
        // 전송 실패 시 경고 메시지
        QMessageBox::warning(this, "전송 오류", "메시지 전송 실패");
    }
}
```

이 부분에서는 lineEdit에 작성한 내용을 상대방에게 전송하는 부분입니다. 한글이 잘 들어가도록 utf-8로 인코딩을 하여 작업을 진행하고자 하였고 상대방의 Ip, 상대방의 포트번호를 통해 writeDatagram메서드로 데이터를 전송하는 부분입니다.

```cpp
// 메시지 수신 처리
void MainWindow::readyRead()
{
    // 받을 데이터가 있는 동안 반복
    while (udpSocket->hasPendingDatagrams()) {
        QByteArray buffer;
        buffer.resize(udpSocket->pendingDatagramSize());

        QHostAddress sender;
        quint16 senderPort;

        // 데이터그램 읽기
        udpSocket->readDatagram(buffer.data(), buffer.size(), &sender, &senderPort);

        // 받은 메시지를 문자열로 변환
        QString message = QString::fromUtf8(buffer);
        QString currentTime = QTime::currentTime().toString("hh:mm:ss");

        // 채팅창에 받은 메시지 표시
        ui->chatDisplay->append(QString("[%1] 상대방: %2").arg(currentTime, message));

        // 채팅창을 가장 아래로 스크롤
        ui->chatDisplay->verticalScrollBar()->setValue(
            ui->chatDisplay->verticalScrollBar()->maximum());
    }
}
```

이 부분은 readyRead메서드로, QUdpSocket에서 readyRead 시그널이 발생하면 이에 대한 슬롯메서드로 호출되게 한 것입니다. 예제에 있는 코드 및 원리를 그대로 사용하고자 했습니다. 데이터를 받을 때에는 pendingDatagramSize로 버퍼로부터 받을 데이터의 크기를 미리 알고 resize하여 buffer의 크기를 키운 뒤에 그 후에 받아야 합니다. 받는 사람인 자신에 대한 IP주소와 , 해당 포트 번호를 기입하여 데이터를 받는 부분입니다. 이번에도 받은 메시지를 한글로 정확히 받을 수 있도록 버퍼로부터 받은 값을

utf-8로 변환하여 출력하도록 했습니다.