3.1 트랜스포트 계층 서비스 및 개요

🔑 트랜스포트 계층 프로토콜은 각기 다른 호스트에서 동작하는 애플리케이션 프로세스간의 논리적 통신을 제공한다.
🔑 논리적 통신은 애플리케이션 관점에서 보면 프로세스들이 동작하는 호스트들이 직접 연결된 것 처럼 보인다.
🔑 송신측은 송신 애플리케이션 프로세스로부터 수신한 메시지를 트랜스포트 계층 세그먼트라는 트랜스포트 계층 패킷으로 변환한다.
🔑 이 변환은 애플리케이션 메시지를 세그먼트로 만들기 위해 작은 조각으로 분할하고 각각의 조각에 트랜스포트 계층 헤더를 추가함으로써 수행된다.

3.1.1 트랜스포트 계층과 네트워크 계층 사이의 관계

🔑 트랜스포트 계층 프로토콜은 각기 다른 호스트에서 동작하는 프로세스들 사이의 논리적 통신을 제공하지만, 네트워크 계층 프로토콜은 호스트르들 사이의 논리적 통신을 제공한다.
🔑 트랜스포트 계층 프로토콜은 종단 시스템에 존재한다.
🔑 종단 시스템안에서 트랜스포트 프로토콜은 애플리케이션 프로세스에서 네트워크 경계(즉, 네트워크 계층)까지 메시지를 운반하며, 또한 반대 방향으로 네트워크 계층에서 애플리케이션 프로세스로 메시지를 운반한다.
🔑 컴퓨터 네트워크는 애플리케이션에서 서로다른 서비스를 제공하도록 하는 개별 프로토콜을 갖는 다양한 트랜스포트 프로토콜을 만들 수 있다.
🔑 트랜스포트 계층이 제공할 수 있는 서비스는 하위 네트워크 계층 프로토콜의 서비스 모델에 의해 제약 받는다.
🔑 만약 네트워크 계층 프로토콜이 호스트 사이에서 전송되는 트랜스포트 계층 세그먼트에 대한 지연 보장이나 대역폭 보장을 제공할 수 없다면, 트랜스포트 계층 프로토콜은 프로세스끼리 전송하는 메시지에 대한 지연 보장이나 대역폭 보장을 제공할 수 없다.
🔑 그럼에도 불구하고 하위 네트워크 프로토콜이 상응하는 서비스를 제공하지 못할 때도, 특정 서비스는 트랜스포트 프로토콜에 의해 제공될 수 있다.
🔑 네트워크 프로토콜이 패킷을 분실하거나 손상시키거나 복사본을 만들 때도, 애플리케이션에게 신뢰적인 데이터 전송 서비스를 제공할 수 있다.
🔑 네트워크 계층이 세그먼트의 기밀성을 보장할 수 없을 때도 침입자가 애플리케이션 메시지를 읽지 못하도록 암호화를 사용할 수 있다.

3.1.2 인터넷 트랜스포트 계층의 개요

🔑 UDP: 비신뢰적이고 비연결형인 서비스를 요청한 애플리케이션에게 제공한다.
🔑 TCP: 신뢰적이고 연결지향형 서비스를 요청한 애플리케이션에게 제공한다.
🔑 애플리케이션 개발자는 두 가지 프로토콜 중 하나를 명시하여야 한다.
🔑 UDP 패킷을 데이터그램으로 표현하기도 한다.
🔑 인터넷의 네트워크 계층 프로토콜은 인터넷 프로토콜(IP, Internet Protocol)라는 이름을 갖는다.
🔑 IP 서비스 모델은 호스트들 간에 논리적 통신을 제공하는 최선형 전달 서비스이다.
🔑 이것은 IP가 통신하는 호스트들 간에 세그먼트를 전달하기 위해 최대한 노력하지만, 어떤 보장도 하지 않는다는 것을 의미한다.
🔑 또한 무결성(세그먼트의 전달 및 순서대로 전달을 보장)을 보장하지 않는다. 따라서 IP는 비신뢰적인 서비스이다.
🔑 UDP와 TCP의 가장 기본적인 기능은 종단 시스템 사이의 IP전달 서비스를 종단 시스템에서 동작하는 두 프

로세스간의 전달 서비스로 확장하는 것이다.
🔑 호스트 대 호스트 전달을 프로세스 대 프로세스 전달로 확장하는 것을 트랜스포트 계층 **다중화**와 **역다중화**라고 부른다.
🔑 UDP와 TCP는 헤더에 오류 검출 필드를 포함함으로써 무결성 검사를 제공한다.
🔑 이러한 최소한의 두 가지 서비스가 UDP가 제공하는 유일한 두 가지 서비스이다.
🔑 UDP는 하나의 프로세스에 의해 전송된 데이터가 손상되지 않고 목적지 프로세스에 도착한다는 것을 보장하지 않는다.
🔑 반면 TCP는 신뢰적인 데이터전송을 제공한다. 흐름제어, 순서 번호, 확인 응답, 타이머를 사용함으로써 TCP는 송신하는 프로세스로부터 수신하는 프로세스에게 데이터가 순서대로 정확하게 전달되도록 확실하게한다.
🔑 이처럼 TCP는 종단시스템간에 IP의 비신뢰적인 서비스를 프로세스 사이의 신뢰적인 데이터전송 서비스로 만들어준다.
🔑 또한 TCP는 혼잡 제어를 사용한다. 이것은 통상적인 서비스처럼 야기한 애플리케이션에게 제공되는 특정 서비스가 아니라 전체를 위한 일반 서비스이다.
🔑 한 TCP 연결이 과도학 양의 트래픽으로 모든 통신하는 호스트들 사이의 스위치와 링크를 혼잡하게 하는 것을 방지하는 것이 TCP 혼잡제어이다.
🔑 TCP는 혼잡한 네트워크 링크에서 각 TCP연결이 링크의 대역폭을 공평하게 공유하여 통과하도록 해준다. 이것은 송신측의 TCP가 네트워크에 보낼 수 있는 트래픽을 조절함으로써 수행된다.
🔑 반면 UDP 트래픽은 조절되지 않는다. UDP 프로토콜을 사용하는 애플리케이션은 허용이 되는 한 그것이 만족하는 어떤 속도로든 전송할 수 있다.

3.2 다중화와 역다중화

🔑 호스트 대 호스트 전달 서비스에서 호스트에서 동작하는 애플리케이션에 대한 프로세스 대 프로세스 전달 서비스로 확장하는 것을 살펴본다.
🔑 목적지 호스트에서의 트랜스포트 계층은 바로 아래의 네트워크 계층으로부터 세그먼트를 수신한다.
🔑 트랜스포트 계층은 한 호스트에서 동작하는 해당 애플리케이션 프로세스에게 이 세그먼트의 데이터를 전달하는 의무를 진다.
🔑 프로세스가 소켓을 갖고 있다는 사실을 우리는 배웠다.
🔑 트랜스포트 계층은 실제로 데이터를 직접 프로세스로 전달하지 않는다. 대신에 중간 매개자인 소켓에게 전달한다.
🔑 어떤 주어진 시간에 수신측 호스트에 하나 이상의 소켓이 있을 수 있으므로 각각의 소켓은 하나의 유일한 식별자를 갖는다. 이 식별자의 포맷은 소켓이 UDP인지 TCP인지에 따라 달라진다.
🔑 이제 수신측 호스트가 수신한 트랜스포트 계층 세그먼트를 어떻게 적절한 소켓으로 향하게 하는지를 생각해보자.
🔑 각각의 트랜스포트 계층은 수신 소켓을 식별하기 위해 세그먼트에 필드 집합을 가지고 있다.
🔑 수신측의 트랜스포트 계층은 수신 소켓을 식별하기 위해 이러한 필드를 검사한다. 그리고 이 세그먼트를 해당 소켓으로 보낸다.
🔑 트랜스포트 계층 세그먼트의 데이터를 올바른 소켓으로 전달하는 작업을 **역다중화**(demultiplexing)라고 한다. 🔑 출발지 호스트에서 소켓으로부터 데이터를 모으고, 이에 대한 세그먼트를 생성하기 위해 각 데이터에 헤더 정보(역다중화에 사용)로 캡슐화하고, 그 세그먼트들을 네트워크 계층으로 전달하는 작업을 **다중화**라고 한다.
🔑 중간 호스트의 트랜스포트 계층은 네트워크 계층 아래로부터 수신한 세그먼트를 위쪽의 프로세스 p_1 또는 p_2로 반드시 역다중화해야 한다.
🔑 이것은 도착한 세그먼트의 데이터가 이에 상응하는 프로세스의 소켓으로 전달되도록해서 이루어진다.
🔑 또한 중간 호스트의 트랜스포트 계층은 프로세스의 소켓으로부터 외부로 나가는 데이터를 모으고, 다음엔 트랜스포트 계층 세그먼트들로 만들고, 이 세그먼트들을 아래 네트워크 계층으로 전달해야만 한다.

🔎 비록 인터넷 트랜스포트 프로토콜의 내용에서 다중화와 역다중화를 설명했지만, 이들은 한 계층에서의 한 프로토콜이 그 상위계층의 여러 프로토콜에 의해 사용될 때 마다 관련된 것임을 알아두기 바란다.
🔎 트랜스포트 계층 다중화에는 두 가지 요구사항이 있다.

1. 출발지 포트 번호 필드
2. 목적지 포트 번호 필드

🔎 각각의 포트 번호는 0~65535 까지의 16비트 정수이다. 그 중에서 0~1023까지의 포트 번호를 잘 알려진 포트번호라고 하며 사용을 엄격하게 제한하고 있다. (RFC 1700에 명시)
🔎 호스트의 각 소켓은 포트 번호를 할당받는다. 그리고 세그먼트가 호스트에 도착하면 트랜스포트 계층은 세그먼트안의 목적지 포트번호를 검사하고 상응하는 소켓으로 세그먼트를 보내게 된다. 그러면 세그먼트의 데이터는 소켓을 통해 해당되는 프로세스로 전달된다.
🔎 이것은 UDP의 기본적인 작동 방식이며 TCP의 다중화/역다중화는 좀 더 많은 의미를 갖고 있음을 볼 것이다.

비연결형 다중화와 역다중화

```
clientSocket = socket(AF_INEF, SOCK_DGRAM)
```

🔎 이 방법으로 트랜스포트 계층은 포트번호를 현재 호스트에서 UDP로 사용하지 않는 1024~65535 사이의 포트번호를 자동으로 할당 한다.

```
clientSocket.bind(("",19157))
```

🔎 소켓을 생성한 뒤에 소켓에 소켓bind()를 사용하여 특정 포트번호를 UDP소켓에 할당할 수 있다.
🔎 UDP소켓 19157을 가진 호스트 A의 프로세스가 호스트 B의 UDP소켓 46428을 가진 프로세스에게 애플리케이션 데이터 전송을 원한다고 가정하자.
🔎 호스트 A의 트랜스포트 계층은 애플리케이션 데이터, 출발지 포트번호(19157), 목적지 포트번호(46428), 그리고 2개의 다른 값을 포함하는 트랜스포트 계층 세그먼트를 생성한다.
🔎 트랜스포트 계층은 만들어진 세그먼트를 네트워크 계층으로 전달한다. 네트워크 계층은 세그먼트를 IP 데이터그램으로 캡슐화하고 최선형 전달 서비스로 세그먼트를 수신호스트로 전달한다.
🔎 이 세그먼트가 수신 호스트 B에 도착하면 수신 호스트는 세그먼트 안의 목적지 포트번호(46428)를 검사하고 그 세그먼트를 적절한 소켓으로 보낸다.(역다중화)
🔎 UDP 소켓이 목적지 IP주소와 목적지 포트번호로 구성된 두 요소로 된 집합에 의해 식별된다는 것을 이해하자.
🔎 출발지 IP주소와 출발지 포트번호가 모두 다르거나 어느 하나만 다를지라도 같은 목적지 IP주소와 목적지 포트번호를 가지면 2개의 세그먼트는 같은 목적지 소켓을 통해 같은 프로세스로 향할 것이다.
🔎 이제 출발지 포트번호는 무슨 목적으로 사용되는지 궁금할지도 모른다.
🔎 A에서 B로가는 세그먼트에서 출발지 포트 번호는 '회신 주소'의 한 부분으로 사용된다. 즉, B가 세그먼트를 다시 A에게 보내기를 원할 때, B에서 A로 가는 세그먼트의 목적지 포트번호는 A로부터 B로가는 세그먼트의 출발지 포트번호로부터 가져온다.

연결지향형 다중화와 역다중화

🔎 TCP소켓과 UDP소켓의 다른 점은 TCP소켓은 4개 요소의 집합, 즉 출발지 IP, 출발지 포트번호, 목적지 IP, 목적지 포트번호에 의해 식별된다는 것이다.
🔎 특히, UDP와는 다르게 다른 출발지 주소 또는 다른 출발지 포트번호를 가지고 도착하는 2개의 TCP 세그먼트(초기 연결 설정 세그먼트 제외)는 2개의 다른 소켓으로 향하게 된다.

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, 12000))
```

🔎 TCP 서버 애플리케이션은 '환영 소켓'을 가지고 있다. 이 소켓은 포트번호 12000을 가진 TCP 클라이언트로부터 연결 설정 요청을 기다린다.
🔎 TCP 클라이언트는 위의 명령으로 소켓을 생성하고 연결 설정 요청 세그먼트를 보낸다. 연결 설정 요청은 목적지 포트 번호 12000(서버의 환영소켓)과 TCP 헤더에 설정된 특별한 연결설정 비트를 가진 TCP 세그먼트이다.

```
connectionSocket, addr = serverSocket.accept()
```

🔎 서버 프로세스로 동작하는 컴퓨터의 호스트 OS가 목적지 포트번호 12000을 포함하는 연결요청 세그먼트를 수신하면, 이 세그먼트를 포트 번호 12000으로 연결 수락을 기다리는 서버 프로세스로 보낸다.
🔎 서버는 연결 요청 세그먼트의 다음과 같은 네가지 값에 주목한다. 1. 출발지 포트번호, 2. 출발지 IP주소, 3. 목적지 포트번호, 4. 목적지 IP주소 새롭게 생성된 연결 소켓은 이 네가지 값에 의해 식별된다.
🔎 그 다음에 도착하는 모든 세그먼트의 출발지 포트, 출발지 IP, 목적지 포트, 목적지 IP의 값들이 일치하면 세그먼트는 이 소켓으로 역다중화될 것이다.
🔎 호스트 C가 서버 B로 2개의 HTTP 세션을 시작하고, 호스트 A가 호스트 B로 하나의 HTTP 세션을 시작한다고 가정하자.
🔎 호스트 A, C와 서버 B는 각자 유일한 IP주소 A, C, B를 갖고 있고 호스트 C는 2개의 출발지 포트번호 26145, 7532를 자신의 HTTP 연결에 할당한다.
🔎 호스트 A는 C와 독립적으로 출발지 포트 번호를 선택하므로, 이것 또한 HTTP 연결에 출발지 포트번호로 26145를 할당할 수 있다. 그렇게 하더라도 2개의 연결은 다른 출발지 IP주소를 가지므로 서버 B는 같은 출발지 포트 번호를 가진 두 연결을 여전히 올바르게 역다중화 할 수 있다.

웹 서버와 TCP

🔎 아파치 웹 서버(Apache Web Server)같은 웹 서버가 포트 번호 80상에서 동작하는 호스트를 고려해보자.
🔎 클라이언트가 서버로 세그먼트를 보내면, 오는 세그먼트는 목적지 포트 번호 80을 갖고 있을 것이다. 서버는 각기 다른 클라이언트가 보낸 세그먼트를 출발지IP주소와 출발지 포트 번호로 구별한다.
🔎 그러나 연결 소켓과 프로세스 사이에 항상 일대일 대응이 이루어지는 것은 아니다. 실제로 오늘날의 많은 고성능 웹 서버는 하나의 프로세스만을 사용한다. 그러면서 각각의 새로운 클라이언트 연결을 위해 새로운 연결 소켓과 함께 새로운 스레드(가벼운 서브 프로세스)를 생성한다.
🔎 하나의 같은 프로세스에 붙어 있지 않은 연결 소켓들(다른 식별자를 가진) 이 동시에 존재할 수 있다.
🔎 만약 클라이언트와 서버가 지속적인 HTTP를 사용한다면, 지속적인 연결의 종속 기간에 클라이언트와 서버는 같은 서버 소켓을 통해 HTTP메시지를 교환할 것이다.
🔎 그러나 많약 클라이언트와 서버가 비지속적인 HTTP를 사용한다면, 모든 요청 및 응답 마다 새로운 TCP 연결이 생성되고 종료될 것이다. 이러한 빈번하게 발생하는 소켓 생성과 종료는 바쁘게 일하는 웹 서버 성능에 심각한 부담을 준다.

3.3 비연결형 트랜스포트:UDP

🔎 적어도 트랜스포트 계층은 네트워크 계층과 해당하는 애플리케이션 레벨 프로세스간의 데이터를 넘겨주기 위해 다중화와 역다중화 서비스를 제공해야 한다.
🔎 UDP는 다중화/역다중화 기능과 간단한 오류 검사 기능을 제외하면 IP에 아무것도 추가하지 않는다. UDP를 선택한다면 거의 IP와 직접 통신하는 셈이다.
🔎 UDP는 세그먼트를 송신하기 전에 송신 트랜스포트 계층 개체들과 수신 트랜스포트 계층 개체들 사이에 핸드셰이크를 사용하지 않는 비연결형이다.
🔎 DNS는 전형적으로 UDP를 사용하는 애플리케이션 프로토콜의 예이다. 호스트의 DNS 애플리케이션이 질의를 생성할 때, DNS 질의 메시지를 작성하고 UDP에게 메시지를 넘겨준다.
🔎 목적지 종단 시스템에서 동작하는 UDP 개체와 호스트측 UDP는 어떠한 핸드셰이크도 수행하지 않고 메시지에 헤더 필드를 추가한 후에 최종 세그먼트를 네트워크 계층에 넘겨준다. 네트워크 계층은 UDP 세그먼트를 데이터그램으로 캡슐화하고 네임서버에 데이터그램을 송신한다.
🔎 이때 질의 호스트에서의 DNS 애플리케이션은 질의에 대한 응답을 기다린다. 만약 질의 호스트가 응답을 수신하지 못한다면 질의를 다른 네임서버로 송신하거나 이를 야기한 애플리케이션에게 응답을 수신할 수 없음을 통보한다.
🔎 왜 애플리케이션 개발자가 TCP보다 UDP방식으로 애플리케이션을 개발하려고 하는지 궁금할 것이다.
🔎 TCP는 신뢰적인 데이터 전송 서비스를 제공하지만 UDP는 그렇지 않으므로 TCP가 항상 더 선호될까? 그렇지 않다. 왜냐하면 많은 애플리케이션은 다음과 같은 이유로 UDP에 더 적합하다.
🔎 무슨 데이터를 언제 보낼지에 대해 애플리케이션 레벨에서 더 정교한 제어

- UDP는 데이터를 UDP 세그먼트로 만들고, 그 세그먼트를 즉시 네트워크 계층으로 전달한다. 이에 반해서 TCP는 혼잡제어 매커니즘을 가지고 있다.
- 이 혼잡제어 매커니즘은 목적지 호스트들과 출발지 호스트들 사이에서 하나 이상의 링크가 과도하게 혼잡해지면, 트랜스포트 계층 TCP송신자를 제어한다.
- 또한 TCP는 신뢰적인 전달이 얼마나 오래 걸리는지에 관계 없이 목적지가 세그먼트의 수신 여부를 확인 응답할 때 까지 데이터의 세그먼트 재전송을 계속할 것이다.
- 실시간 애플리케이션은 최소 전송률을 요구할 때도 있고, 지나치게 지연되는 세그먼트 전송을 원하지 않으며, 조금의 데이터 손실을 허용할 수도 있으므로, TCP의 서비스 모델은 이러한 애플리케이션 요구와는 맞지않다.
- 이러한 애플리케이션은 UDP를 사용하고, 애플리케이션의 한 부분으로서 UDP의 기본 세그먼트 전달 외에 필요한 어떤 추가 기능을 구현할 수 있다.

🔎 연결 설정이 없다.

- TCP는 데이터 전송을 시작하기전에 세 방향 핸드셰이크를 사용한다. 반면에 UDP는 공식적인 사전준비 없이 전송한다. 그러므로 UDP는 연결을 설정하기 위한 어떤 지연도 없다. 이것은 DNS가 왜 TCP보다는 UDP에서 동작하는지에 대한 일반적인 이유이다. 만약 DNS가 TCP에서 동작한다면 많이 느려질 것이다.
- HTTP 문서로된 웹 페이지는 신뢰성이 중요하기 때문에 UDP보다는 TCP를 사용한다. 그러나 HTTP에서 TCP연결 설정 지연은 웹 문서 다운로드 지연의 심각한 원인이다.
- 실제로 크롬 브라우저에서 사용되는 QUIC(Quick UDP Internet Connection)프로토콜은 기본 트랜스포트 프로토콜로 UDP를 사용하고 UDP위에 애플리케이션 계층 프로토콜의 안정성을 구현한다.

🔎 연결 상태가 없다.

- TCP는 종단 시스템에서 연결 상태를 유지한다. 이 연결상태는 수신 버퍼와 송신버퍼, 혼잡제어 파라미터, 순서 번호와 확인응답 번호 파라미터를 포함한다.

- 이에 반하여 UDP는 연결상태를 유지하지 않으며 이 파라미터 중 어떤 것도 기록하지 않는다. 이러한 이유로 일반적으로 특정 애플리케이션 전용 서버는 애플리케이션 프로그램이 TCP보다 좀 더 많은 액티브 클라이언트를 수용할 수 있다.

🔑 작은 패킷 오버헤드

- TCP는 세그먼트마다 20바이트의 오버헤드를 갖지만, UDP는 단지 8바이트 오버헤드를 갖는다.

애플리케이션	애플리케이션 계층 프로토콜	하위 트랜스포트 프로토콜
전자메일	SMTP	TCP
원격 터미널	텔넷	TCP
보안 원격 터미널 접속	SSH	TCP
웹	HTTP, HTTP/3	TCP(HTTP용), UDP(HTTP/3용)
파일 전송	FTP	TCP
원격 파일 서버	NFS	일반적으로 UDP
스트리밍 멀티미디어	DASH	TCP
인터넷 폰	통상 독점 프로토콜	UDP 또는 TCP
네트워크 관리	SNMP	일반적으로 UDP
이름변환	DNS	일반적으로 UDP

🔑 전자 메일, 원격 터미널 전송, 웹, TCP상의 파일 전송 등의 애플리케이션은 TCP의 신뢰적인 데이터 전송이 필요하다.
🔑 2장의 초기 버전의 HTTP는 TCP를 통해 실행되었지만 최신 버전의 HTTP는 UDP를 통해 실행되어 애플리케이션 계층에서 자체 오류 제어 및 혼잡제어를 제공한다.
🔑 그럼에도 불구하고 많은 중요한 애플리케이션들이 TCP보다 UDP에서 동작한ㄷ. 예를 들면 UDP는 네트워크 관리(SNMP) 데이터를 전달하는데 사용된다. 네트워크 관리 애플리케이션은 네트워크가 혼잡한 상태에 있을 때 자주 동작해야 하므로 이러한 경우에는 UDP가 TCP보다 더 바람직하다. (과도한 상태에서, 신뢰적이거나 혼잡제어된 데이터 전송은 수행하기 어렵다.)
🔑 DNS는 TCP의 연결설정 지연을 피하려고 UDP에서 동작한다. 표에서 볼 수 있듯이 UDP와 TCP둘 다 인터넷 전화, 실시간 비디오 회의, 저장된 오디오와 비디오의 스트리밍 같은 멀티미디어 애플리케이션에 종종 사용된다.
🔑 이러한 모든 애플리케이션에서는 적은 양의 패킷 손실은 허용할 수 있다. 그래서 신뢰적인 데이터 전송이 애플리케이션의 성능에 절대적으로 중요한 것만은 아니다. 게다가 인터넷 전화와 화상 회의 같은 실시간 애플리케이션들은 TCP 혼잡제어가 나쁜영향을 미친다. 그래서 개발자들은 이 경우 UDP에서 동작하도록 구현한다.
🔑 패킷 손실률이 낮고, 보안 측면의 이유로 UDP 트래픽을 막는 일부 기관에서 TCP는 점점 더 스트리밍 매체 전송에 매력적인 프로토콜이 되고 있다.
🔑 오늘날 멀티미디어 애플리케이션을 UDP위에서 동작시키는 방법이 일반적으로 사용되지만 아직 논의의 여지가 있다.
🔑 UDP는 혼잡 제어를 제공하지 않는다. 그러나 혼잡제어는 네트워크가 꼭 필요한 작업을 할 수 없게 되는 폭주 상태에 빠지는 것을 막기 위해 반드시 필요하다. 만약 모두가 혼잡 제어를 사용하지 않고 높은 비트의 비디오 스트리밍을 시작한다면, 라우터에 많은 오버플로가 발생할 것이고, 이는 소수의 UDP 패킷만이 출발지-목적지 간의 경로를 무사히 통과할 것이다.

🔎 또한 무엇보다도 제어되지 않은 UDP송신자에 의해 발생한 높은 손실률은 그 손실률을 감소시키기 위해 TCP송신자들이 속도를 줄이도록 할 것 이다.
🔎 그러므로 UDP의 혼잡제어 결여는 UDP송신자 간의 높은 손실률을 초래할 수도 있고, TCP세션의 혼잡이 발생할 수 있으며 이는 잠재적으로 심각한 문제점이다.
🔎 UDP를 사용할 때도 신뢰적인 데이터 전송이 가능하다는 것에 주목하자. 만약 애플리케이션이 신뢰성을 애플리케이션 자체에서 제공한다면 신뢰적인 데이터 전송이 가능하다.
🔎 QUIC 프로토콜은 UDP상에서 애플리케이션 계층에 신뢰성을 구현했다고 언급했다. 그러나 이것은 애플리케이션 개발자가 오랜시간 분주하게 디버깅하게 하는 어려운 작업이다.
🔎 그런데도 신뢰성을 애플리케이션에 직접 포함하는 것은 애플리케이션이 '동시에 두마리 토끼를 잡는' 격이다. 즉, 애플리케이션 프로세스들은 TCP의 혼잡제어 매커니즘에 의해 전송률 억제를 강요당하지 않고도 신뢰적으로 통신할 수 있다.

3.3.1 UDP 세그먼트 구조

🔎 애플리케이션 데이터는 UDP데이터그램의 데이터 필드에 위치한다. 예를 들면, DNS에 대한 데이터 필드에는 질의 메시지나 응답 메시지를 포함한다. 스트리밍 오디오 애플리케이션에는 오디오 샘플이 데이터 필드를 채운다.
🔎 UDP헤더는 2바이트씩 구성된 단 4개의 필드만을 갖는다. 출발지 포트번호, 목적지 포트번호를 가지며 목적지 호스트가 목적지 종단 시스템에서 동작하는 정확한 프로세스에게 애플리케이션 데이터를 넘기게 해준다.
🔎 체크섬(checksum)은 세그먼트에 오류가 발생했는지를 검사하기 위해 수신 호스트가 사용한다. 사실 체크섬은 UDP세그먼트 이외에 IP헤더의 일부 필드도 계산한다.

UDP세그먼트 32bit

출발지 포트 번호	**목적지 포트 번호**
길이	체크섬

애플리케이션 데이터(메시지)

3.3.2 UDP 체크섬

🔎 UDP 체크섬은 오류 검출을 위한 것이다. 즉, 체크섬은 세그먼트가 출발지로부터 목적지로 이동했을 때, UDP 세그먼트안의 비트에 대한 변경사항이 있는지 검사하는 것이다.
🔎 송신자 측에서 UDP는 세그먼트 안에 있는 모든 16비트 워드의 합산에 대해 다시 1의 보수를 수행하며, 합산 과정에서 발생하는 오버플로는 윤회식 자리올림을 한다. 이 결괏값이 UDP세그먼트의 체크섬 필드에 삽입된다.
🔎 체크섬 계산의 간단한 예를 살펴보자 다음과 같은 3개의 16비트 워드가 있다고 하자.

0110011001100000
0101010101010101
1000111100001100
이러한 16비트 워드에서 처음 2개의 워드 합은 다음과 같다.
0110011001100000
0101010101010101
1011101110110101
앞의 계산의 합에 세번째 워드를 더하면 다음과 같은 결과가 나온다.
1011101110110101
1000111100001100
1 0100101011000001

오버플로된 값을 첫번째 자리에 더해준다. (윤회식 자리올림)
0100101011000010

🔎 마지막 합은 오버플로가 있고 이를 윤회식 자리올림을 했음에 유의하자. 1의 보수는 모든 0을 1로 변환하고 모든 1을 0으로 변환하면 구할 수 있다. 그래서 합의 1의 보수는 1011010100111101이고, 이것이 체크섬이 된다.
🔎 수신자에서는 체크섬을 포함한 4개의 모든 비트의 16비트 워드들이 더해진다. 만약 패킷에 어떤 오류도 없다면, 수신자에서의 합은 1111111111111111이 될 것이다. 만약 비트들 중에 하나라도 0이 있다면 패킷에 오류가 발생했음을 알 수 있다.
🔎 여러분들은 많은 링크 계층 프로토콜이 오류검사를 제공하는데 왜 UDP가 체크섬을 제공하는지 궁금할 것이다. 그 이유는 출발지와 목적지 사이의 모든 링크가 오류 검사를 제공한다는 보장이 없기 때문이다. 즉, 링크 중에서 하나가 오류 검사를 제공하지 않는 프로토콜을 사용할 수도 있는 것이다.
🔎 그러므로 세그먼트들이 정확하게 링크를 통해 전송되었을지라도, 세그먼트가 라우터의 메모리에 저장될 때 비트 오류가 발생할 수가 있다. 주어진 링크 간의 신뢰성과 메모리의 오류 검사가 보장되지 않고, 종단감의 데이터 전송 서비스가 오류검사를 제공해야 한다면, UDP는 종단 기반으로 트팬스포트계층에서 오류 검사를 제공해야만 한다. 이것이 시스템 설계에서 그 유명한 종단과 종단의 원칙(end-end priciple)의 한 예이다.
🔎 IP는 어떠한 2계층 프로토콜에서도 동작해야 하므로 트랜스포트 계층은 안전장치로서 오류검사를 제공하는 것이 유용하다. UDP는 오류검사를 제공하지만, 오류를 회복하기 위한 어떤 일도 하지 않는다.
🔎 일부 UDP 구현에서는 손상된 세그먼트를 그냥 버리기도 하고, 다른 구현에서는 경고와 함께 손상된 세그먼트를 애플리케이션에게 넘겨주기도 한다.