Ch4. Transport Protocol

Created	@Jun 16, 2020 5:51 PM
Tags	컴망

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/240b05b8-399d-4ac0-8ae1-9b419abf0265/Chapter_4_Transport_Protocol.pptx

트랜스포트 계층
트랜스포트 계층 vs. 네트워크 계층
포트 번호, 소켓 주소
Multiplexing/Demultiplexing
    Demultiplexing은 어떻게 작동하는가?
Connectionless, Connection-oriented
UDP: User Datagram Protocol
    UDP: Segment header
신뢰성 있는 Data transfer
    rdt1.0
    rdt2.0
    rdt2.1
    rdt2.2 : a NAK-free protocol
    rdt3.0 : channels with errors and loss
Stop-and-wait 방법
    Utilization 계산
Pipelining 방법
    Go-back-N
    Selective Repeat
TCP : Overview
TCP Segment Structure
    TCP 헤더
TCP RTT, RTO (Retransmission TimeOut)
    Estimated RTT
    Dev RTT (Safety margin)
    그래서 결론 : Timeout Interval
TCP : Reliable data transfer
TCP Fast retransmit
TCP Flow control

Connection Management
    Handshaking : 3-way handshake
    Connection closing
TCP 전송률
Congestion control
    Pipelined control
    TCP Slow Start
    Loss가 발생하면?
TCP Throughput
TCP Fairness
SCTP (Stream Control Transmission Protocol)
DCCP (Datagram Congestion Control Protocol)
    TCP-like congestion control
    TCP-friendly rate control

트랜스포트 계층

서로 다른 호스트에서 돌아가는 프로세스 사이의 커뮤니케이션 제공

- 트랜스포트 프로토콜은 **end system에서 돌아간다. end-to-end 개념이다.**
  (중간에 지나다니는 라우터들은 TL을 가지지 않는다. TL은 말단의 호스트들만 가진다.)
    - 송신 측 : 앱의 메시지를 Segment로 나눠서, 필요한 헤더를 붙여서, NL에 전달.
        - **Encapsulation** : AL의 메시지 (→ Payload)에 헤더를 붙여 TL에서 Packet이 되는 과정.
    - 수신 측 : 받은 Segment들을 다시 모아서 메시지로 만들어서, 앱 레이어에 전달.
        - **Decapsulation** : TL에서 받은 Packet에서 헤더를 떼서 메시지로 만들어 AL로 올리는 과정.
- **TCP**, **UDP** 등이 있다.
    - **TCP** : 신뢰성 O, 순서 O
    - **UDP** : 신뢰성 X, 순서 X

💡 이런 서비스들은 안 된다!
- **Delay guarantees** : '반드시 5초 안에 전달되어야 한다'
- **Bandwidth guarantees** : '최소한 이 속도로 전달되어야 한다'
인터넷은 보장형 서비스가 아니다. Best-effort 서비스이

트랜스포트 계층 vs. 네트워크 계층

- **Network Layer** : **호스트 간 커뮤니케이션 제공**
- **Transport Layer** : 서로 다른 호스트 내부의 **프로세스 간 커뮤니케이션 제공**

포트 번호, 소켓 주소

트랜스포트 계층에서 받은 메시지를 → 애플리케이션 계층의 올바른 프로세스에 전달해줘야 한다.

- **프로세스를 어떻게 구분하는가? → 포트 번호.**
  - 포트 번호의 존재의의 : Demultiplexing을 통해 프로세스를 구분하기 위함.
- **소켓 주소 = IP주소 + 포트 번호**

Multiplexing/Demultiplexing

- **송신 측**에서는 **Multiplexing**을 한다.
  - TL 헤더에 소켓 정보 저장
- **수신 측**에서는 **Demultiplexing**을 한다.
  - 수신한 Segment들을 올바른 소켓에 전달하기 위해 헤더 정보 사용

Demultiplexing은 어떻게 작동하는가?

- 호스트가 받는 IP Datagram의 구조
  - 각 **Datagram**은 **Source IP주소, Dest IP주소**를 가진다.
  - **Datagram 하나당 TL Segment 하나**를 가진다.
  - 각 **Segment**는 **Source 포트 번호, Dest 포트 번호**를 가진다.
- 호스트는 IP주소와 포트 번호를 사용해 → 적절한 소켓에 Segment를 전달한다.

Connectionless, Connection-oriented

TL은 두 종류의 서비스를 제공한다.

- **Connectionless service : UDP**
  - 연결 과정 없이 바로 메시지를 보낸다.
  - 메시지 순서가 바뀔 수 있다.
- **Connection-oriented service : TCP**
  - 초기 연결이 성립되어야 전송이 가능하다.
  - [0, 1, 2] 보냈는데 2가 먼저 가면? 2는 hold되고 1이 마저 와야 그제서야 2가 도달한다.

TCP Demultiplexing은 Source IP주소, Source 포트 번호, Dest IP주소, Dest 포트 번호 4개를 모두 사용한다.

UDP: User Datagram Protocol

- 각 UDP Segment는 독립적으로 핸들링된다.
- UDP를 쓰는 경우
  - 스트리밍 - loss가 조금 있어도 되지만, 속도는 빨라야 한다.
    - TCP는 UDP에 비해 느리다.
    - [1, 2, ..., 8] 보냈을 때, [1, 2, 8]이 오게 되면 3, 4, 5, 6, 7 다 올 때까지 8이 buffer에서 hold된다.
- UDP에 신뢰성을 더하려면?
  - AL에서 별도 처리 필요 (에러 복구)

UDP: Segment header

- **Source 포트 번호** (16bit)
- **Dest 포트 번호** (16bit)

- **길이** (16bit)
- **체크섬 (에러 판별)** (16bit)
  - UDP Segment에 에러가 있으면 어떻게 하는가? → Drop한다.

신뢰성 있는 Data transfer

rdt (reliable data transfer) 프로토콜을 만들어보자.

rdt1.0

Reliable하다면, bit error도, packet loss도 없다.

그냥 보내면 된다. 그냥 받으면 된다.

rdt2.0

하위 레이어에서 **bit flip이 발생할 수 있다.** 어떻게 하면 좋은가?

- **Error detection**이 먼저 되어야 한다.
  - **Checksum**을 이용해 가능.
- 그 다음은, **Error recover**도 되어야 한다.
  - **수신 측에서 잘 받았으면 ACK, 에러가 있으면 NAK**를 송신 측에 보낸다.
  - 송신 측은 **NAK를 받았으면** 에러로 간주하고, **해당 패킷을 다시 보낸다.**

그러나, 큰 문제가 있다.

- **ACK/NAK 자체에 error가 발생하면?**
  - 송신 측에서는 받은 이 데이터가 ACK인지 NAK인지 모른다. 수신 측에서 무슨 일이 일어났는지 모른다.
  - 그냥 데이터를 재전송해버리면? 수신 측에서는 잘 받은 건데... 수신 측은 중복된 데이터를 받게 된다.
  - → **패킷에 Sequence number를 추가**해서, 이전에 받은 데이터인지 확인하면 된다.

rdt2.1

Sequence number는 0과 1만 있어도 충분하다.

rdt2.2 : a NAK-free protocol

메시지 종류가 많아지면, 메시지 종류를 구분하기 위한 비트 필드 길이도 길어진다. → 메시지 종류를 줄이자.

- **NAK을 따로 두지 않고**, 송신 측에서는 **중복되는 ACK가 들어오면 오류로 간주**하고, 패킷을 다시 보낸다.

rdt3.0 : channels with errors and loss

지금까지는 error만 봤고, loss가 생기면 어떻게 하는가? loss 발생 시에는 ACK도 무용지물이다.

- ACK를 기다리는 대기 시간을 정해두고, **기다려도 ACK이 안 오면 (timeout), 패킷을 다시 보낸다.**
  - timeout을 너무 짧게 잡으면, 수신 측에서 중복된 데이터를 받는 문제가 있다.
  - → Remind: 중복된 데이터는 **Sequence number**로 해결 가능.

Stop-and-wait 방법

지금까지의 방법이 바로 **Stop-and-wait 방법**이다.
보내고, ACK 오기 전까지 기다리고, ACK 오면 다음거 또 보내고, ...

너무 비효율적이다. 그냥 보내는 대로 보내고, ACK 받는 대로 받으면 안 될까?

Utilization 계산

L : 데이터의 크기, R : Rate (전송 속도), RTT : Round-Trip Time (왕복 시간) 일 때, Utilization 계산

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{\text{데이터 전송에만 걸리는 시간}}{\text{전체 걸리는 시간}}$$

Pipelining 방법

여러 개를 연속으로 보낸다. 막 보내고, ACK도 막 온다.

연속으로 보낼 수 있는 패킷의 최대 개수가 정해져 있다. **ACK가 안 온 상태로 최대 N개까지만 보낼 수 있다. (N = Window size)**

💡 L 크기의 데이터를 3개 연속으로 보내면, Utilization 3배.

Pipelining에는 두 방식이 있다.

Go-back-N

가장 오래 기다린 패킷이 timeout되면, 그 때 같이 보냈던, 아직까지 못 받은 애들 모두 재전송.

- **cumulative ACK** : 7번까지 보낸 거 잘 받았다! *(ACK #N : N-1번까지 잘 받았다. 이제 N번 보내줄 차례다.)*
- **Buffer가 필요없다.** 버퍼링이 필요없다. 한번 못 받은 패킷이 있으면 그 뒤로 받는 패킷들은 모두 버린다.

Selective Repeat

각각의 패킷에 타이머를 달아서, **timeout된 패킷만 재전송한다.**

- **individual ACK** : 1번 잘 받았다, 2번 잘 받았다, ... *(ACK #N : N번 잘 받았다)*
- **Buffer를 사용한다.**
- (주의) Window size와 Sequence number를 잘 설정해야 딜레마에 빠지지 않는다.

TCP : Overview

- **Point-to-point.** 두 명이서 일대일 통신.
- **Reliable.**
- **In-order byte stream.** 소켓마다 버퍼가 있다. 메시지가 들어오면, TCP는 바이트 단위로 순서대로 저장한다. 응용 단에서는 알아서 가져가면 된다. 바이트가 버퍼에 연속으로 주르륵 도착하는 것. Message boundary가 없다.
  - UDP : 메시지가 오면 메시지 하나가 단위가 된다. 나뉘어지지 않는다. Message boundary가 있다.
- **Pipelined.** 한 번에 많은 패킷이 왔다갔다하니까, **Congestion control(혼잡 고려)**, **Flow control(수신측의 버퍼 상황 고려)** 기능을 지원한다.
- **Full duplex.**
- **Connection-oriented.** 커넥션을 먼저 만드는 Handshaking을 먼저 거친다.

TCP Segment Structure

TCP 헤더

- **Source 포트 번호 (16bit)**
- **Dest 포트 번호 (16bit)**

> 💡 포트 번호 0~65535 (2^16 = 65536)

> 💡 왜? Multiplexing/Demultiplexing 위해.
> 프로세스별로 소켓을 가지고, 포트 번호를 가진다. 소켓을 구분하기 위해 포트 번호가 필요하다.

- **Sequence number (32bit)** : 세그먼트 개수가 아니라, **데이터의 바이트마다 카운팅**한다.
- **ACK number (32bit)** : 마찬가지로 바이트 카운팅.
  - cumulative ACK와 마찬가지로, **ACK #N : N-1까지 잘 받았다. 이제 N번 받을 차례다.**
- **HeadLen (4bit)** : 뒤에 나올 옵션에 따라 헤더 길이가 달라지기 때문
- not used **(6bit)**
- **UAPRSF (각 1bit, 총 6bit)**
  - **URG** : 급한 데이터 (Urgent data)
  - **ACK** : 1이면 ACK valid
  - **PSH** : 커널 버퍼에 임계치를 넘어서까지 데이터를 푸시해라
  - **RST** : 1이면 Connection을 리셋하자는 거구나!
  - **SYN** : 1이면 Connection을 맺자는 거구나!
  - **FIN** : 1이면 Connection을 끊자는 거구나!
- **Receive window (16bit)** : Flow control에 사용 **(송신 측의 수신버퍼 남은 크기를 담는다)**

- **Checksum (16bit)** : 헤더뿐만 아니라 데이터 Segment 전체에 대한 Checksum
- **Urg data pointer (16bit)** : URG이 1일 때 유효. 급한 데이터의 마지막 바이트 위치.
- **옵션 (가변 길이)** : 송신 시각 등 필요한 정보

TCP RTT, RTO (Retransmission TimeOut)

Timeout(RTO)을 어떻게 설정할 것인가?

→ **RTT 평균값** 구해서, 이를 기반으로 RTO를 설정하면 되겠다.

Estimated RTT

Exponential weighted moving average 방식

$$EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \times SampleRTT$$

EstimatedRTT : 지금까지 구한 평균
SampleRTT : 현재 측정한 RTT값
가중치 α가 의미하는 것 : 가중치가 크면, 현재 측정한 값에 가중치를 더 주겠다는 것
가중치는 보통 $\alpha = 0.125$로 설정

Dev RTT (Safety margin)

Deviation(진폭)에 대한 평균도 구하자.

$$DevRTT = (1-\beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$$

보통 $\beta = 0.25$로 설정

그래서 결론 : Timeout Interval

$$TimeoutInterval = EstimatedRTT + 4 \times DevRTT$$

TCP : Reliable data transfer

TCP는 NL (IP계층) 위에서 rdt 서비스를 만들어준다.

> 💡 **패킷 Timeout (RTO)**이나, **중복된 ACK**가 들어오면 재전송한다.

TCP Fast retransmit

RTO까지 기다리기에는 오랜 시간이 걸린다.

→ 전송 중 일부 Segment가 누락된 채로 계속 송신하면, 중복된 ACK를 계속 받게 된다.

→ ∵ Cumulative ACK

→ 중복된 ACK를 통해 빠르게 Lost segment를 Detect하자.

> 💡 TCP Fast retransmit : 중복 ACK를 3회 받으면, ACK로 받은 가장 작은 Sequence number의 Segment를 보낸다.

TCP Flow control

> 💡 **RcvBuffer** : 소켓 버퍼의 사이즈
> **rwnd : Free buffer space**

Remind: TCP 송신 측은 **헤더의 Receive window 공간에 자신의 남은 소켓 버퍼 사이즈 (rwnd)**를 담는다.
→ **버퍼가 넘치지 않게 Flow control!**

Connection Management

TCP는 데이터 전송 전에 **Handshake : 초기 연결 설정**을 한다.

Handshaking : 3-way handshake

1. **A → B : SYNbit = 1, seq = x**
   (SYN할 거고, 내 seq는 x다)
2. **A ← B : SYNbit = 1, seq = y, ACKbit = 1, ACKnum = x + 1**
   (잘 받았고, SYN할 거고, 내 seq는 y다)
3. **A → B : ACKbit = 1, ACKnum = y + 1**
   (잘 받았다)

> 💡 seq는 각자 랜덤한 번호로 생성

Connection closing

1. **A → B : FINbit = 1, seq = x**
   (FIN할 거고, 내 seq는 x다)
2. **A ← B : ACKbit = 1, ACKnum = x + 1**
   (잘 받았다)
3. **A ← B : FINbit = 1, seq = y**
   (나도 FIN할 거고, 내 seq는 y다)
4. **A → B : ACKbit = 1, ACKnum = y + 1**
   (잘 받았다)

> 💡 A가 3번 작업 후 기다릴 때, RTO보다 2배 더 기다린다.

TCP 전송률

$$rate \approx \frac{cwnd}{RTT}\ bytes/sec$$

Congestion control

Congestion(혼잡) = 너무 많은 소스가, 너무 많은 데이터를, 너무 빨리 보내서, 핸들링이 안 되는 것.

💡 Flow control과는 다르다.

- **Lost packet** (라우터에서 버퍼 오버플로우) = 송신 측에서 ACK를 영원히 못 받는다.
- **Long delay** (라우터 버퍼에 Queueing) = 송신 측에서 RTO가 일어날 수 있다.

Pipelined control

cwnd(congesion window size)를 동적으로 조절한다.

💡 cwnd : ACK를 받지 않은 상태에서 최대 연속으로 보낼 수 있는 패킷 양. (≈ Pipeline에서의 window size)

네트워크 상황이 좋아지면 → **Additive Increase. 매 RTT마다 1 MSS씩조금씩 증가.**

- 네트워크 상황이 나빠지면 → **Multiplicative Decrease. 한번에 절반 감소.**

cwnd 바이트만큼 보내고, RTT만큼 기다리고, 또 보낸다.

TCP Slow Start

첫 전송에 cwnd는 1 MSS이고, 이후 **매 RTT마다 cwnd를 2배씩 증가**시킨다.

- 첫 전송 : 1 segment, 두번째 전송 : 2 segments, 이후 4 segments, 8 segments, ...씩 보내진다.

그러나 이대로 두면, 보내는 양이 엄청나게 커질 것이다.

→ **Slow Start 단계**에서는 이렇게 cwnd를 **Exponential하게 증가**시키고, **일정 수준 이후에는 Congestion Avoidance 단계라 해서 cwnd를 선형으로 증가시킨다.**

Loss가 발생하면?

- Congestion으로 인해 **Timeout이 발생하면?**
  - 다시 **cwnd를 1 MSS로 설정하고 반복**한다.
- Congestion으로 인해 **ACK 3회 중복되면?**
  - **cwnd를 반으로 줄이고 다시 반복**한다.

TCP Reno : ACK 3회 중복 시 반으로 줄임 *(ACK 3번 오는 건 timeout보다는 상태가 좋다는 거니까)*
TCP Tahoe : Timeout, ACK 3회 중복, 두 경우 모두 무조건 1로 설정
(TCP Vegas : 측정된 RTT를 보고 적당히 조절.)

TCP Throughput

W : window size

$$avg\ window\ size = \frac{3}{4} \times W\ bytes$$

$$avg\ TCP\ throughput = \frac{3}{4} \times \frac{W}{RTT}\ bytes/sec$$

L : Loss 확률

$$TCP\ throughput = \frac{1.22 \times MSS}{RTT\sqrt{L}}\ bytes/sec$$

오히려 Throughput 10Gbps를 달성하기 위해 L을 계산하는 문제도 나올 수 있다.

TCP Fairness

여러 커넥션이 대역폭 R을 공유한다면, **커넥션끼리 공평하게 R을 나눠가져야 한다. → 각 Connection이 $\frac{R}{K}$만큼 R을 나눠갖도록.**

→ TCP는 **Additive Increase, Multiplicative Decrease를 통해 대역폭을 Fair하게 유지할 수 있다.**

UDP도 같이 쓰는 방식으로 Fairness를 확보할 수도 있다.
라우터에서는 TCP용 버퍼, UDP용 버퍼를 따로 둔다.

하나의 애플리케이션이 여러 병렬 Connection을 열 수도 있다.
$\frac{R}{Connection개수}$ 하면 된다.

SCTP (Stream Control Transmission Protocol)

TL에 TCP, UDP만 있는 건 아니고, 여러 개 있다.

Multihoming, Multiple-stream.

DCCP (Datagram Congestion Control Protocol)

UDP지만, **Connection-oriented**로 작동하고, **가볍고**, **Congestion Control 지원**.

두 가지 방식으로 나눌 수 있다.

TCP-like congestion control

ACK 사용. 근데 데이터 빠졌다고 재전송하지 않음

TCP-friendly rate control

공식에 의해 한번에 보내는 rate를 control함