컴네 기말

▼ rdt 3.0 성능

- Stop-and-wait방식이다.
- U_{sender} = 사용률을 계산해보자.
  - 송신측이 사용하는 시간의 비율이다.
- 예를 들어
  - 1 Gbps link, 15 ms propagation delay, 8000 bit packet에 대해
  - 채널에 패킷을 보내는데 걸리는 시간
    - transmission delay = $D_{trans} = L/R$ = 8 microsec 정도가 나온다.
  - 송신측은 채널에 패킷을 전송하기만 하면, 응답이 돌아올 때 까지는 아무것도 하지 않는다.
  - = RTT(응답이 돌아오는데 걸린 시간) + D_{trans} (수신측이 ACK를 채널에 올리는데 걸린시간)
  - 그렇다면 U_{sender} 값은
  - $U_{sender} = (D_{trans})/(D_{trans} + RTT) = (L/R)/(L/R + RTT)$
  - 대략 0.00027이 나온다.
- 이렇게 rdt 3.0의 성능은 개구리다.
  - 프로토콜 때문에 그 밑에 깔린 채널의 성능이 제대로 발휘되지 못하고 제한된다.
- Pipelining
  - 송신측이 패킷을 다수 송신할 수 있게 만든다.
  - 즉 ACK 받지 않은 unACKed(= “in-flight”) 패킷을 여러 개 보내게 하자.
    - 이를 위해선 seq #의 범위가 늘어나야하며, 송수신측이 버퍼링을 진행해야한다.
  - 파이프라이닝을 사용했을 경우, 송신측 사용율을 올릴 수 있다.
    - 3개의 패킷을 보낼때, 3배의 사용율이 나온다.
  - 파이프라이닝을 수행하는 방법은 크게 두가지로 나뉜다.

- Go-Back-N (GBN)
  - 수신자가 특정 패킷을 받지 못하면, 그 패킷부터 시작하는 모든 패킷을 다시 보낸다.
    - 정상적으로 도착한 패킷도 다시 보낸다.
  - 송신측은 window 를 만든다.
    - 윈도우는 패킷을 송신할 범위를 결정한다.
    - 윈도우의 범위가 결정되면, 그 범위 내에 있는 seq # 의 패킷은 모두 송신된다.
    - ACK 받은 패킷만큼 윈도우를 옮겨 새로운 패킷을 보낼 수 있게 만든다.
    - 타임 아웃되면, 윈도우의 처음부터 다시 다 보낸다.
  - 수신측은 window 만큼 보내진 패킷을 연속으로 수신한다.
    - cumulative ACK
    - 수신한 각 패킷을 누적시키며 연속적인 ACK를 보낸다.
      - 여기까지는 내가 잘 받았음
    - 만약 패킷이 손실되면, 이 누적된 패킷 사이에 빈 틈이 생긴다
    - 빈 틈이 생기기 직전 = 연속이 끊기기 직전 수신한 패킷에 대한 ACK를 반복적으로 보낸다.
      - 이 마지막 번호는 rcv_base에 저장된다.
  - 장단
    - 구현이 단순, 관리하기가 편하다.
    - 하나의 윈도우에 하나의 타이머로 충분하다.
    - 대량의 재전송이 발생할 수 있으며, 중복된 패킷이 전송될 수 있다.
- Selective repeat
  - 수신자가 수신된 모든 패킷 각각에 ACK를 보내는 방식이다.
  - 송신 측은 마찬가지로 윈도우를 가진다.
    - 윈도우 내부의 모든 패킷에 대해 ACK를 기다린다.
    - 각각의 패킷에는 타임아웃이 있으며, 타임아웃이 지나면 그 패킷을 재전송한다.

- 윈도우 내부의 가장 작은 seq # 의 패킷이 ACK되면, 그제서야 윈도우를 한칸 옮긴다.

- 수신 측은 패킷을 저장하는 버퍼와 윈도우를 가진다.
  - 송신된 패킷을 버퍼에다 저장하고, ACK를 보낸다.
    - 순서가 어긋난 경우 버퍼에 저장해놓고,
    - 순서가 맞으면 deliver한다.
  - 손실된 패킷을 재전송받으면 버퍼에 넣고 순서대로 재조립한다.
- 장단.
  - 패킷 손실에 대해 필요한 패킷만을 재전송하므로 효율적이고, 대역폭 낭비가 없다.
  - 적은 양의 재전송으로 충분하다.
  - 구현과 관리가 어려우며, 각 패킷이 각자의 타이머를 가져야한다.
  - 수신 측은 버퍼를 관리해야하므로 메모리 사용량이 증가한다.
- 주의 사항
  - SR의 딜레마!
    - seq #가 0, 1, 2, 3 있다고 가정하고
    - 윈도우 크기가 3이라고 생각하자.
    - 그럼 패킷들은
      - 0, 1, 2, 3, 0 ,1, 2, 3,.... 의 seq #를 가진다.
    - 그럼 윈도우 크기 3이므로
      - 송신측은 0, 1, 2 를 보낸다.
    - 수신 측은 0, 1, 2 패킷을 받는다.
      - 0, 1, 2 ACK를 보내고, 다음 3, 0 , 1 패킷을 받을 준비를 한다.
    - 어 ACK가 오다가 자빠졌다.
      - 송신측은 0, 1, 2 를 다시 보낸다.
      - 수신측은 0, 1을 받았다....?
        - seq #만 같지 다른 패킷이다!

        - 송 수신측이 서로를 알지 못하기에 벌어진 일이다.
    - 딜레마를 해결하기 위해서
        - 윈도우의 크기는 (max seq #) / 2를 넘어가선 안된다!
        - 전체 ACK가 자빠지더라도, 한 바퀴 돌아 0으로 시작하지 않는다.
- Sun 마이크로 시스템.

▼ TCP / IP

- TCP 세그먼트와 IP 데이터그램(IPv4)
    - 각각 20바이트를 가진다.
    - TCP는 source / dest 포트 번호(16비트)를 가지고
    - IP는 source / dest IP 주소(32비트)를 가진다
- IPv6 IP 데이터그램은
    - Priority를 가진다.
    - IP 주소가 128 비트로 늘어났다.
    - IPv4와 비교해서
        - checksum 삭제 (라우터에서 빠른 처리를 위해서)
        - 파편화와 재조립 삭제
        - 기타 옵션 삭제.
- TCP 통신은 ACKs를 기반으로 이루어진다.
    - 패킷은 seq#를 가진다.
    - ACK이란 다음에 받길 '기대, 예상하는' 패킷의 seq #가 된다.
        - Cumulative ACK
            - 받은 패킷에 대해서 ACK를 누적하여 증가시킨다.
            - 여러 개의 패킷을 받고, 앞의 패킷에 대한 ACK가 자빠진다고 해도 ACK는 누적되어 있으므로 불필요한 재전송을 막는다
            - 하나의 ACK로 여러 세그먼트를 처리하는 장점이 있다.
        - 내가 수신한 패킷의 seq # + (데이터 사이즈)을 다음에 받길 예상한다

        - seq# = 0 일 때, 데이터 사이즈가 0 바이트면 무한 반복이 일어나므로, ACK = 1로 보낸다.
    - 그럼 수신자는 순서가 어긋난 세그먼트는 어떻게 처리하나요?
        - TCP 프로토콜이 알 바가 아니다. 어플리케이션 구현한 사람이 알아서 해라.
    - 가상의 통신을 예로 들어보자.
        - 데이터의 크기는 1바이트다.
        - seq = 42 패킷, ACK = 79를 보냈다.
            - 난 seq = 79인 패킷을 받길 기대한다.
        - seq = 42인 패킷과 ACK = 79를 받았으므로
            - ACK에 맞는 seq = 79인 패킷을 보낸다.
            - 그런데 내가 받은게 seq = 42니까, 다음 seq = 43 패킷을 기대한다.
            - ACK = 43을 보낸다.
        - seq = 79를 받고, ACK = 43을 받았으니
            - ACK에 맞는 응답을 보낸다 seq = 43
            - seq = 79 다음 패킷을 기대한다 ACK = 80
- 그럼 TCP의 RTT를 계산해보자.
    - 왜?
        - 응답이 올 때 까지 기다리는 타임 아웃을 '적절히' 정하기 위해서
            - 최소한 RTT보다는 긴 값을 가져야한다!
        - 너무 짧으면
            - ACK가 잘 오고 있는데 불필요한 재전송이 일어난다.
            - Premature timeout(섣부른 타임아웃)
        - 너무 길면?
            - 세그먼트 손실에 대해 반응이 너무 느리다.
    - 그럼 RTT를 측정하는 방법은?
        - SampleRTT
            - ACK를 수신할 때 까지 걸린 시간(재전송은 무시!) = 지금의RTT

        - EstimatedRTT
            - 지금까지 측정한 RTT들의 평균이다.
        - RTT는 계속해서 변화하므로,,,,
            - 이번에 보낼 세그먼트의 RTT를 갱신해주어야 한다.

            > 💡 $EstimatedRTT = (1-\alpha)EstimatedRTT + \alpha SampleRTT$

            - 일반적인 경우, 알파값은 0.125를 쓴다.
                - 알파가 높아질 수록 = 최근 측정한 RTT에 조금 더 가중치를 두겠다.
                - exponential weighted moving average (EWMA)
                    - = 과거 측정한 값의 영향력은 급속히 낮아진다!
    - 최종적인 타임아웃 값은
        - 예상RTT + 안전 마진을 더한다.
            - TimeoutInterval = EstimatedRTT + 4 * DevRTT
        - 이때 DevRTT는 오차의 평균이다.
            - $DevRTT = (1-\beta)DevRTT + \beta|SampleRTT - EstimatedRTT|$
            - 일반적으로 베타는 0.25를 사용한다.

▼ TCP MSS

- Maximum Segment Size
- 최대 보낼 수 있는 데이터 패킷의 크기
    - 기본은 536바이트
    - 대부분의 경우 MSS의 크기는 OS가 결정해 준다.
- TCP handshake를 할때
    - TCP SYN 패킷을 통해 상호간 MSS의 크기를 결정한다.
        - 송수신 양 측의 MSS 중 작은 것으로 합의한다.

    - 한 번 결정되면, 연결 중에는 변경할 수 없다.
- 그럼 MSS의 크기는?
    - 가장 큰 IP 데이터그램의 크기와 동일하게 MSS를 설정한다.
        - 즉 보낼 수 있는 가장 큰 데이터(MTU)를 설정하고, 거기서 TCP와 IP의 헤더(각 20바이트)를 제외한다.
        - MTU = 576 바이트일때 (기본 IP 데이터그램 사이즈)
        - MSS = 576 - (20 + 20) = 기본 TCP 세그먼트 사이즈
    - 왜요?
        - 만약 이거보다 큰 경우, 세그먼트가 전송되는 도중 분할(fragmentation) 될 수 있다.
        - 따라서 불필요한 재전송, 추가적인 오버헤드와 지연이 생길 수 있다.

▼ TCP congestion contorl

- 혼잡 제어?
    - 네트워크의 패킷이 과도하게 증가하는 경우 → 혼잡하다.
    - 혼잡해진 네트워크에선 패킷 손실이 발생하고, 손실된 패킷을 또 재전송하고.... 또 손실나고..... 네트워크 전체가 자빠지게 된다.
    - 따라서 네트워크의 처리 속도가 따라갈 수 있을 만큼만 패킷을 송신해라!
        - 패킷 손실이 생겼으면 속도를 좀 늦춰라!
- AIMD
    - 가장 기본적인 혼잡 제어
    - AI
        - Additive Increase
        - loss가 감지될 때까지 송신율을 점진적으로 증가시킴.
    - MD
        - Multiplicative Decrease
        - loss가 발생할 때 마다 송신률을 반 토막 줄임.
    - 최종적으로 Saw Tooth모양 ⇒ 톱니 모양의 송신률이 나오게 된다.
        - 대역폭을 탐침(probing)한다.

- TCP slow start
  - 점진적인 증가를 지수적인 증가로 진행하자.
  - Exponential하게 → AI 에서 MI로 가듯이.
  - 초기 송신율은 느리나(세그먼트 한 개)
    - cwnd = congestion window = 1 MSS(최대 세그먼트 크기)
  - 송신율이 빠르게 증가한다(2개 → 4개 → 8개....)
    - 1 RTT가 지날 때 마다 cwnd를 2배로 올린다.
  - 무한대로 빠르게 증가하는건 아님.(= congestion avoidance)
    - 변수 ssthresh = threshold가 있고, 이 위로는 linear하게 증가시킴.
      - cwnd ++
    - ssthresh의 값은 얼마로 두는게 좋겠니?
      - 첫 값은 OS가 정해주고,
      - 다음 부터는 loss가 발생했을 때의 전송률 / 2로 정한다.
        - ex) 12에서 loss가 발생했으면, 다음 ssthresh는 6이 된다.
      - 따라서 ssthresh는 가변적으로 변동한다.
  - 그럼 패킷 로스가 발생하면?
    - TCP Tahoe
      - 전송률을 아예 초기화, 1부터 다시 시작하자
      - cwnd ⇒ 1 MSS로 되돌리자.
        - Tahoe는 일반적으로 타임아웃으로 로스가 발생한 경우가 된다.
    - TCP Reno
      - 아예 초기화하는건 너무 손실이 크다.
      - ssthresh부터 전송률을 다시 시작하는건 어때.
        - cwnd ⇒ ssthresh로 시작하자.
      - 현재 대부분의 디바이스가 이 방식을 사용한다.
        - 중복 ACK 3개가 쌓이면 (똑같은 ACK를 4개 받았을때)
- TCP fast retransmit
  - 빠른 재전송.

    - 다수 패킷을 전송할 때, 하나의 패킷이 loss된다면,
        - 그 하나의 패킷에 대한 ACK이 여러 개 날아올 것이다.
        - 그런데, 그 손실된 패킷을 다시 보내려면 타임 아웃까지 기다려야한다! = 시간의 손실이 생긴다.
    - 그럼 타임아웃까지 기다리지 말고, 특정 개수의 ACK를 받으면 바로 재전송을 하자 = fast retransmit
    - 일반적으로 3개의 중복된 ACK를 수신하면 재전송한다.

▼ TCP Flow control

- 흐름 제어
    - 혼잡 제어랑은 완전 다른 이야기다(서로 영향을 미치긴 한다)
    - TCP 프로토콜은 TCP 소켓의 수신 버퍼에다가 데이터를 갖다 놓는다
    - 프로세스는 소켓에서 데이터를 건져간다(택배)
    - 그럼 프로세스가 가져가는 것 보다 빠르게 소켓에 데이터를 쌓아 놓으면? = overflow
    - 소켓에는 적당한 양의 데이터만 넣어놓자...
        - 받을 수 있는 양을 정해서 그 만큼만 수신하자.
- 수신자가 송신자를 통제해서, 송신자가 수신측 버퍼를 오버플로우하지 않게 한다.
    - 너무 빠르게, 너무 많이 보내지 않게 한다.
- rwnd = receive window
    - 버퍼에 남아있는 공간을 알려준다.
    - 이 값을 송신자가 받아가서, 이 값을 넘어가지 않게 조절한다.
- TCP 수신자는 rwnd(남은 버퍼 공간)을 헤더에다가 광고한다.
    - 소켓 생성 시에 수신 버퍼의 크기가 자동 할당되며, 일반적으로 OS가 4096바이트로 설정한다.
- TCP 송신자는 송신하는 데이터(unACKed = in-flight) 양을 ACK에 맞게 제한한다.
- 따라서 수신 버퍼가 넘쳐흐를 일이 없다.

▼ TCP Connection management

- 데이터 교환 전에, 송수신자가 “handshake” 한다.

    - 연결 성립에 합의한다.
    - 연결 매개 변수들을 합의한다.
        - starting seq #s 등...
- 2-way handshake
    - 클라이언트가 req를 보내면 → 서버가 ESTAB
    - 서버가 ESTAB하고 응답해주면 → 클라가 ESTAB
    - 이제 얘기 시작하자.
- 2-way면 충분한게 아닌가?
    - 네트워크에서 2-way가 항상 동작하는가?
        - 딜레이의 변수.
        - 서로 상대방을 볼 수 없다.
            - 클라이언트가 req를 보내놓고 자기가 빤쓰런치면 어떻게 되는가?
            - 즉, 클라이언트는 요청과 응답을 모두 받았으니 송수신이 모두 가능하다는게 확실하다. = 신뢰할 수 있다.
            - 그런데 서버는 요청에 대해 응답을 하긴 했는데, 그게 갔는지는 모른다 = 수신은 확실히 되지만, 송신은 갈지 안갈지 확실치 않다.
            - 서버 입장에서 신뢰할 수 없는 통신이다.
        - 메시지의 손실
            - 재전송된 메시지들 때문에?
        - 메시지 reordering
    - 일반적 상황에선 문제가 없다.
        - req(req1) → req 재전송(req2) → 서버 ESTAB(req1이 도착함) → acc → 클라 ESTAB(acc 도착)
        - 그런데 여기서 클라이언트가 종료되고, 서버도 클라이언트를 잊어버린다.
        - 재전송된 req2에 의해 서버가 다시 ESTAB되고, acc를 클라이언트에 보낸다
        - 클라는 이미 없는데? = half - open된 상태.
- 3-way handshake

    - 이 부분은 어떤 언어에서도 설명되는 부분이 아님(함수 안에서 자연스럽게 동작함)
    - 클라 서버가 일종의 FSM이 된다.
    - 클라이언트
        - SYNbit = 1, Seq = x (initial seq #)
        - LISTEN에서 SYNSENT 상태로 진입
    - 서버
        - SYNbit = 1, Seq = y (initial seq #)
        - ACKbit = 1; ACKnum = x + 1 (x를 수신햇슴다)
        - LISTEN에서 SYN RCVD 상태로 진입
    - 클라이언트
        - ACKbit = 1, ACKnum = y + 1 (y를 수신했슴다)
        - ESTAB
    - 서버
        - ESTAB

▼ 복습

- Wireless vs Wire Network
- 우리는 유선 쪽을 다뤘다.(무선이나 보안 관련은 아예 하지 않았다.)
- 10 year yardstick
    - 무선이 유선 속도를 따라잡는데 일반적으로 10년 정도 걸림.
- 인터넷이 무엇이었나?
- Network edge
    - 리프 노드
- Network core
    - 패킷 / 서킷 스위칭
- 퍼포먼스
    - 로스, 딜레이, 처리율

    - 혼잡 제어 하는 이유? 퍼포먼스. 모든 것의 이유는 퍼포먼스를 올리는 것에 있다.
- 큰 그림을 보시오.
- Leonard Kleinrock → 대단한 저자.
    - 최초의 패킷 스위칭 만들때 도움?
    - vint cerf(TCP/IP 개발)의 스승.
- Application layer
- 소켓
    - Transport layer
- Tim BERNERS-LEE
    - HTTP, Web brower를 만들어 튜링상 수상.
- 챕 2
    - 프로토콜!
    - 애플리케이션 레이어 부분을 중요하게 봐
- 챕3
    - 트랜스포트 레이어 서비스
    - Mux Demux
    - UDP
    - RDT
    - TCP
    - Congestion Control
    - Flow control
    - connection management
- 소켓 관련 해서 나온다.
    - UDP, TCP 소켓이 다룰 수 있는 부분
    - 소켓이 다룰 수 없는 부분.
    - UDP만 가지는 부분
    - TCP만 가지는 부분

    - 둘 다 가지는 부분
- end-to-end
    - 트랜스포트 레이어가 제공하는 기능
    - 끝단 사이의 logical communication
- TCP overview
    - RFC 793, 1122, 2018, 5681, 7323
        - 80년 부터 발전해오고 있는 중
    - Point to Point
        - 송 수신자 각각 하나씩
    - 신뢰 가능하고, 순서가 유지되는 바이트 흐름
    - Full duplex
        - 송 수신이 연결에서 동시에 가능한 것.
        - MSS = 최대 세그먼트 크기.
    - Cumulative ACKs
    - Pipelining
        - 혼잡 제어와 흐름 제어는 window size를 가지고 결정
        - cwnd, rwnd
    - connection-oriented
        - handshake
    - flow controlled
        - 송신자가 수신측 버퍼를 터뜨리면 안되게찌
- UDP seg header
    - source, dest port #
    - length, checksum
- TCP seg
    - seq #
    - ACK #
    - flag bits,

        - A bit = ACK
        - RST, SYN
        - FIN은 어마어마하게 복잡하다. 따라서 하지 않았다.
    - receive window → 오늘 배운거(흐름 제어)
- vint cerf & bob kahn
    - 2004년 TCP 개발로 튜링상 수상.
    - “인터넷의 아버지들”
-