4/6 TCP

Flow Control

각 노드의 Send buffer는 receive buffer가 수용가능한 만큼만 보낸다. 이 정보는 TCP 헤더에 receive buffer size로 담겨서 가고, 계속 서로의 수용 가능한 용량을 리포트함

application에서 `read`하지 않으면 버퍼에 데이터가 쌓이고, 이게 꽉 차면 receive buffer size에 0이 적혀서 리포트될 것. 그걸 받은 노드는 데이터를 보내면 안 되는게 맞음

만약 그런데 sender/receiver가 각각의 역할만 수행한다면 이 경우에 sender는 receiver 입장에서 buffer size가 얼마나 남았는지 알 수 없는 상황이 발생함 (buffer이 읽어서 빈 공간이 생긴 경우)

그래서 sender는 segment에 data를 비우고 주기적으로 보내서 ACK를 받고 receive buffer size를 확인한다.

Connection Management

Connection Establishment — TCP 3-Way Handshake

TCP로 뭔가 하려면 일단 두 노드에 버퍼가 존재해야 하고, 양쪽의 seq#가 세팅되면 된다. (서로의 seq#를 세팅함)

1. Client가 server에 connection 요청
    a. send init seq num (x)
    b. send TCP segment with header(SYN==1, 1bit) msg
2. 서버는 SYNACK를 보내줌
    a. send init seq num (y)
    b. send TCP segment with header(SYN==1, 1bit) (ACK==1, 1bit) msg
    c. ACKNUM == x+1
3. 클라이언트는 ACK를 보냄

    a. ACKNUM == y+1

    b. send TCP segment with header(ACK==1, 1bit) msg

    c. may contain client-to-server data

    d. indicates client is live

3번까지 이루어지지 않으면 receiver는 버퍼를 만들지 않는다.

만약 3번이 없었다면 클라이언트는 반응이 왔으니 ㅇㅋ, 근데 서버 입장에서는 반응이 오지 않았으니 정상적으로 연결되었는지 알 수가 없음.

HTTP 요청 가기 전 두 번의 메시지가, 바로 TCP SYN과 TCP SYNACK인 것임.

Closing Connection

1. 클라이언트가 소켓을 닫음
2. 클라이언트는 FIN을 보냄(닫는다?)
    a. SYN과 같이 header(FIN == 1)인 데이터 없는 세그먼트를 보내는것.
3. 서버는 ACK를 보냄
4. 서버는 FIN을 보냄(닫는다?)
5. 클라이언트는 ACK을 보냄
6. 서버는 이걸 받아서 확인함
7. 이후에도 클라이언트와 서버는 3초 정도를 소켓 아예 닫지 말고 기다릴 것을 권장함
    왜? d ACK가 유실되면, 서버는 계속 c FIN을 보낼텐데 그걸 다시 받을 수 있을 정도의 시간은 대기를 해야하는것.

Congestion Control

개요

Sender는 자료를 보낼 때 다음 두 가지 중 더 상태가 좋지 않은 쪽에 맞춘다

1. Network의 상태, 그 중에서도 가장 병목현상이 발생하는 부분
2. Receiver의 Receive buffer

즉 위 두 가지의 상태를 계속해서 트래킹해야 하는데, 2는 피드백을 해주니 알 수 있지만(flow control) 1은 난해한 측면이 있음.

네트워크는 기본적으로 public하고, 사람들은 한 번에 많은 데이터를 빠르게 주고받기를 원함. 너무 많은 데이터가 네트워크에 돌아다녀서 정체가 이루어지면 사실… 데이터를 더 보내면 안된다.

그런데 TCP는 특성상 네트워크가 막히면 더 많은 세그먼트를 재전송하기 때문에 네트워크 상태를 악화시키고, 그렇게 되면 꿀잼.

따라서 TCP는 네트워크가 막히지 않도록 해야 하며, 그러기 위해 Congestion Control이 필요하다. 즉 TCP는 자기의 통신을 위해 이것을 조절한다. (UDP는 그런 거 없고 그냥 패킷을 쏟아붓는다.)

아무튼 네트워크에 대한 정보를 누가 줄 것이냐에 대한 두 가지 접근 방식이 있고, 1번 방식이 현재 채용하고 있는 방식이다.

1. End-End Congestion ctrl (현실)
    a. 아무런 명시적인 메시지 없음
    b. 내부 상황을 각 endpoint에서 알아서 유추해야 함. 정보도 아무것도 없으니, segment-ACK에 대한 응답시간 정도만 가지고 유추
    c. 한 번에 window size만큼 보냈는데 피드백이 너무 안 오면 window size를 줄이고, 빠르게 나오면 window size를 늘린다. 이게 뭔가 파일을 다운로드받을 때 속도가 계속 변하는 이유
2. Network-Assist Congestion ctrl (이상적)
    a. 라우터가 큐 상황같은 것을 알려줌

3 Main Phases of TCP Congestion Control

네트워크는 공유 자원이고, 한 번 막혔을 때 막힌 걸 푸는 방법은 그냥 다같이 안 보내는 것이라서 막혔다는 것을 감지하면 모든 사람이 다 같이 발을 빼는 식으로 해결하게 된다.

Slow Start

실질적으로 TCP가 처음 연결되었을 때 Congestion control을 위해서, 한 번에 데이터를 쏟아부어보는 것이 아니라 하나씩 보내볼 수밖에 없음. 이 과정에서 데이터 피드백이 잘 오면 두 배씩 window size를 증가시킨다. (exponential) 여기서 하나씩 보낸다고 할 때의 segment window size는 1MSS(maximum segment size)에 해당하는 500Byte다. window size는 MSS단위로 변화한다.

Additive Increase

네트워크가 감당할 수 없을 것 같으면(threshold 초과) exponential → linear로 증가량을 바꿈. window Size를 1MSS씩 증가시킨다.

Multiplicative decrease

Additive Increase 중 packet loss를 탐지하면, 그 순간 window size를 반으로 줄인다.

즉 전송속도는 대략 Congestion WindowSize / Round Trip Time 으로 결정되며 둘 다 동적인 값이지만 con.windowSize가 훨씬 더 변화량이 크고, 결국 이건 네트워크의 영향을 가장 많이 받으므로 네트워크 자체가 매우 중요하단 의미다. 다시 말해 모두의 행동이 모두의 전송속도를 결정하게 된다.

TCP Tahoe(tcp series1), TCP Reno(tcp series 2)

Tahoe는 고전적으로 동작하며, 패킷 유실을 탐지했을 때 다시 Slow start 단계로 내려가며 threshold를 패킷 유실이 탐지된 window size의 절반으로 바꾼다. (1980s)

Reno는 패킷 유실의 메커니즘에 따라 다른 동작을 취한다. 3DACK면 threshold를 패킷 유실이 탐지된 window size의 절반으로 바꾸지만, threshold에서 Additive Increase를 진행하고, Timeout이면 Tahoe와 똑같이 동작한다. (현재)

> 💡 TCP에서 패킷 유실을 timeout이 3DACK로 판단하는데, 이 두 경우가 같은 네트워크 상황인가? 3DACK는 다 잘 갔는데 그것만 안 간 상황이고, 전자는 그 이후 패킷이 다 안 간 거라 후자보다 진짜 정체가 생겼을 가능성이 높음.

TCP Fairness

Congestion Control을 모두가 하고 있을 때, 모두에게 공평한가? 그렇다

웨? Additive increase가 1MSS씩 증가하고, Multiplicative decrease가 반타작을 날리기 때문임. 여러 대의 컴퓨터가 이를 반복하면 중간으로 수렴한다고 한다. 신기…

맹점: 진짜진짜 fair한가? 아님. 한 사람이 TCP Connection을 여러 개 열면 TCP끼리는 괜찮지만 개수만큼 몫이 많아진다.