TCP/IP 흐름제어/혼잡제어

흐름 제어 (Flow Control)

> 송신측과 수신측의 **데이터 처리 속도 차이**를 해결하기 위한 기법

수신 측이 송신 측보다 데이터 처리 속도가 빠르면 문제가 없지만, 송신 측의 속도가 빠를 경우 문제가 생긴다. 수신 측에서 **제한된 저장 용량을 초과한 이후에 도착한 패킷은 손실될 수 있으며**, 만약 손실된다면 불필요한 추가 패킷 전송이 발생하게 된다.

흐름 제어는 위와 같이 송신 측과 수신 측의 **TCP 버퍼 크기 차이**로 인해 생기는 **데이터 처리 속도 차이**를 해결하기 위한 방법이다. 수신 측이 송신자에게 현재 자신의 상태를 피드백 한다.

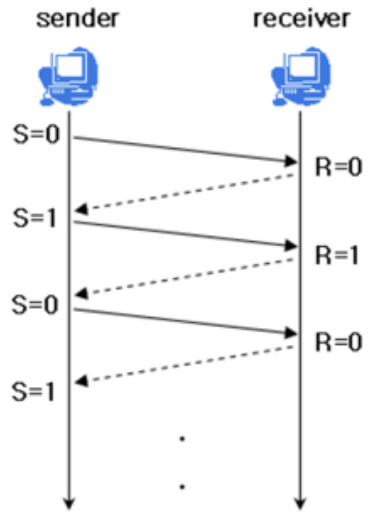
- **TCP 버퍼** : 송신 측은 버퍼에 TCP 세그먼트를 보관한 후 순차적으로 전송하고, 수신 측은 도착한 TCP 세그먼트를 응용이 읽을 때 까지 버퍼에 보관한다.

전송 전체 과정

- 송신 응용 계층 : socket에 데이터를 쓴다.
- 트랜스포트 계층 : 데이터를 세그먼트에 감싸고 네트워크 계층에 넘겨준다.
- 하위 계층 : 수신 노드로 데이터 전송/ 송신자의 send buffer에 데이터 저장, 수신자는 receive buffer에 데이터 저장
- 수신 응용 계층 : 응용에서 준비가 되면 이 buffer에 있는 것 읽기 시작

⇒ **수신 버퍼가 넘치지 않게 하기 위해!**

Stop and Wait

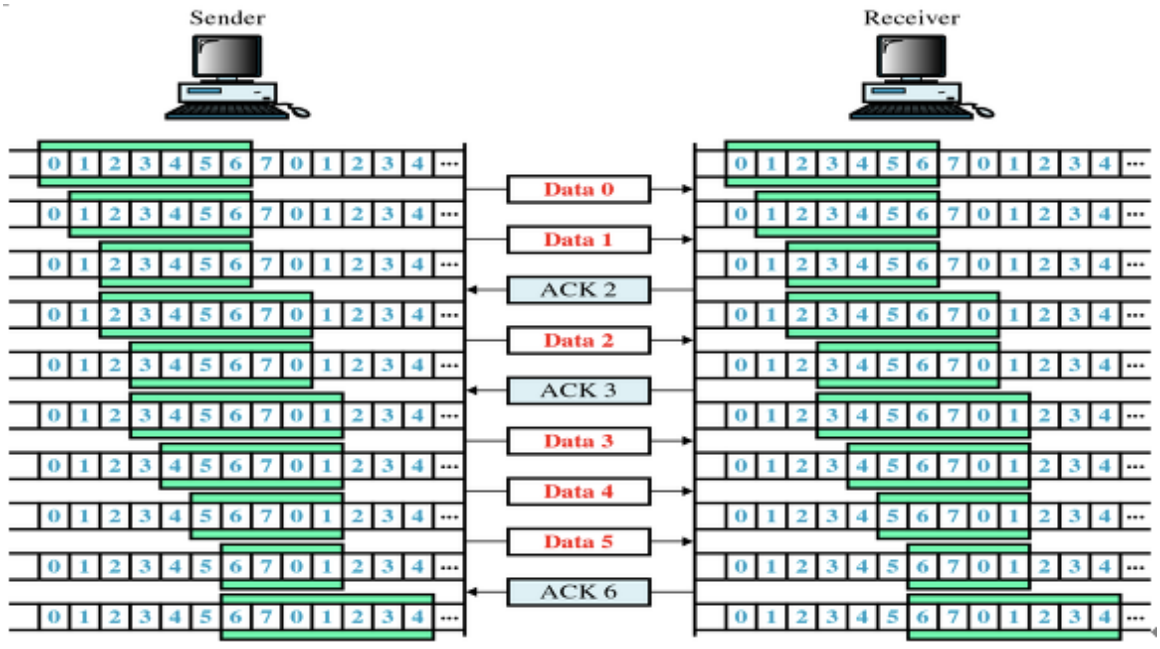


매번 전송한 패킷에 대한 확인 응답을 받아야만 그 다음 패킷을 전송하는 기법이다. 하지만 패킷을 하나씩 보내기 때문에 비효율적인 방법이다.

Sliding Window

수신 측에서 설정한 **윈도우 크기만큼 송신측에서 패킷 각각에 대한 확인 응답 없이 세그먼트를 전송**하게 되고, **데이터 흐름을 동적**으로 조절하는 기법

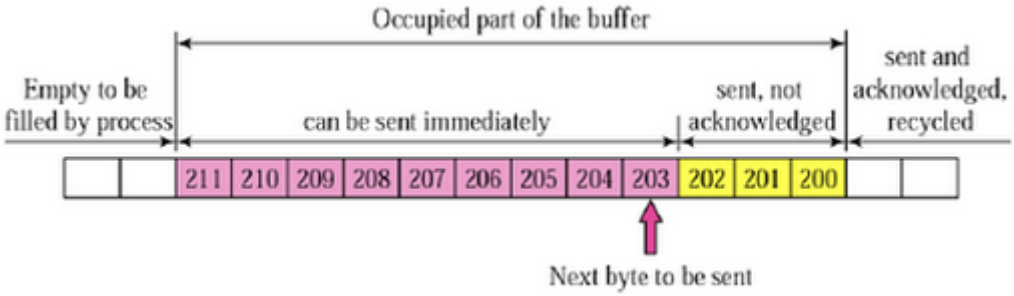
> 전송은 되었지만 ack를 받지못한 바이트의 숫자를 파악하기 위해 사용하는 프로토콜



- 동작방식 : 먼저 **윈도우에 포함되는 모든 패킷을 전송**하고, 그 패킷들의 전달이 확인되는대로 **이 윈도우를 옆으로 옮김으로써 그 다음 패킷들 전송**
- 윈도우 크기 : 최초의 윈도우 크기는 **호스트들의 3-way handshaking을** 통해 수신 측 윈도우 크기로 설정되며, 이후 수신 측의 버퍼에 남아있는 공간에 따라 변한다. 윈도우 크기는 수신 측에서 송신 측으로 **ACK를 보낼 때 TCP 헤더에 담아서 보낸다**. 즉, 윈도우는 메모리 버퍼의 일정 영역이라고 생각하면 된다.

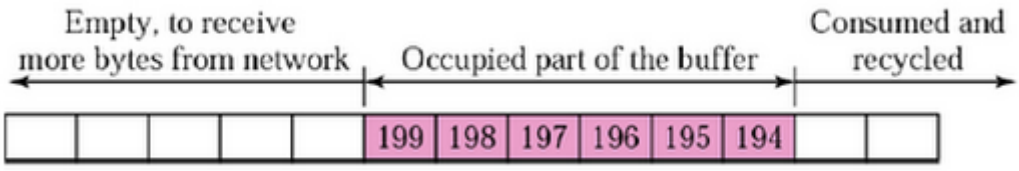
세부 구조

1. 송신 버퍼

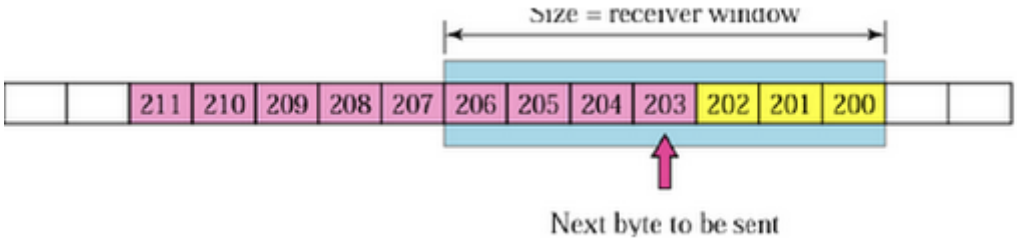


200 전까지는 전송되었고 ACK도 받은 상태이다. 200~202 바이트는 전송 되었으나 ACK를 받지 못한 상태이다. 203~211 바이트는 아직 전송이 되지 않은 상태이다.

2. 수신 윈도우

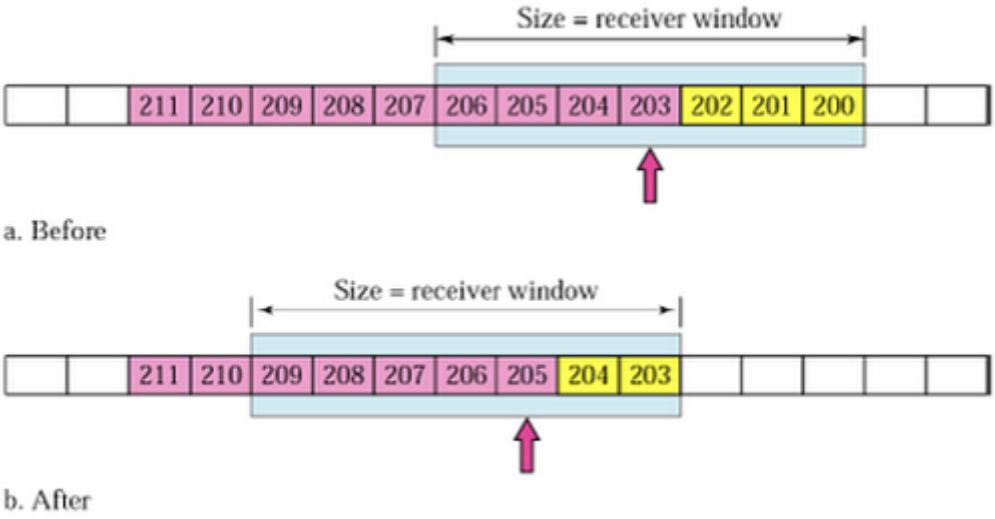


3. 송신 윈도우



수신 윈도우보다 작거나 같은 크기로 지정하게 되면 흐름제어가 가능하다.

4. 송신 윈도우 이동



203~204를 전송하면 수신측에서 ACK 203을 보내고 송신측은 이를 받아 수신 윈도우를 203~209 범위로 이동한다. 205~209가 전송 가능한 상태이다

재전송

송신 측은 일정 시간 동안 수신 측으로부터 ACK를 받지 못하면, **패킷을 재전송**한다. 만약, 송신 측에서 재전송을 했는데 패킷이 소실된 경우가 아니라 수신 측의 버퍼에 남는 공간 없는 경우면 문제가 생긴다. 이를 해결하기 위해 송신 측은 해결 응답(ACK)를 보내면서 **남은 버퍼의 크기(윈도우 크기)**도 함께 보내준다.

혼잡 제어 (Congestion Control)

데이터의 양이 라우터가 처리할 수 있는 양을 초과하면 초과된 데이터는 라우터가 처리하지 못한다. 이때 송신 측에서는 라우터가 처리하지 못한 데이터를 손실 데이터로 간주하고 계속 재전송하여 네트워크를 혼잡하게 한다. 이런 상황은 **송신 측의 전송 속도를 적절히 조절**하여 예방할 수 있는데, 이것을 **혼잡 제어**라고 한다.

흐름 제어는 **송 수신 측 사이의 패킷 수를 제어하는 기능**이라 할 수 있으며, 혼잡 제어는 **네트워크 내의 패킷 수를 조절**하여 네트워크의 오버플로우를 방지하는 기능이다.

AIMD (Additive Increase Multicative Decrease)

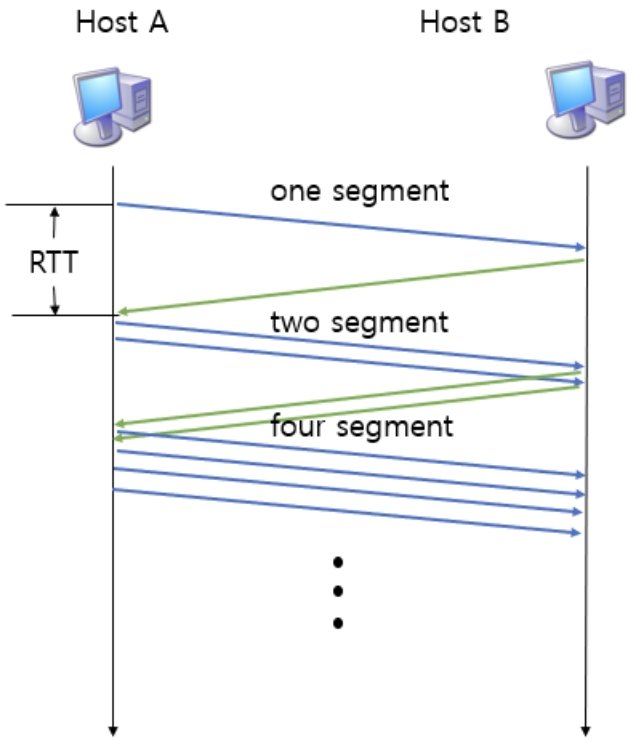
송신 측이 transmission rate(window size)를 **패킷 손실이 일어날 때까지 증가**시키는 접근 방식이다.

- additive increase : 송신 측이 window size를 손실을 감지할 때 까지 매 RTT마다 1 MSS씩 증가시킨다.
- multiplicative decrease : 손실을 감지했다면 송신 측의 window size를 절반으로 감소시킨다.

AIMD는 window size를 1**MSS씩 밖에 증가시키지 않기 때문에** 네트워크의 모든 대역을 활용하여 빠른 속도로 통신하기까지 시간이 오래 걸린다는 단점이 있다.

Slow Start

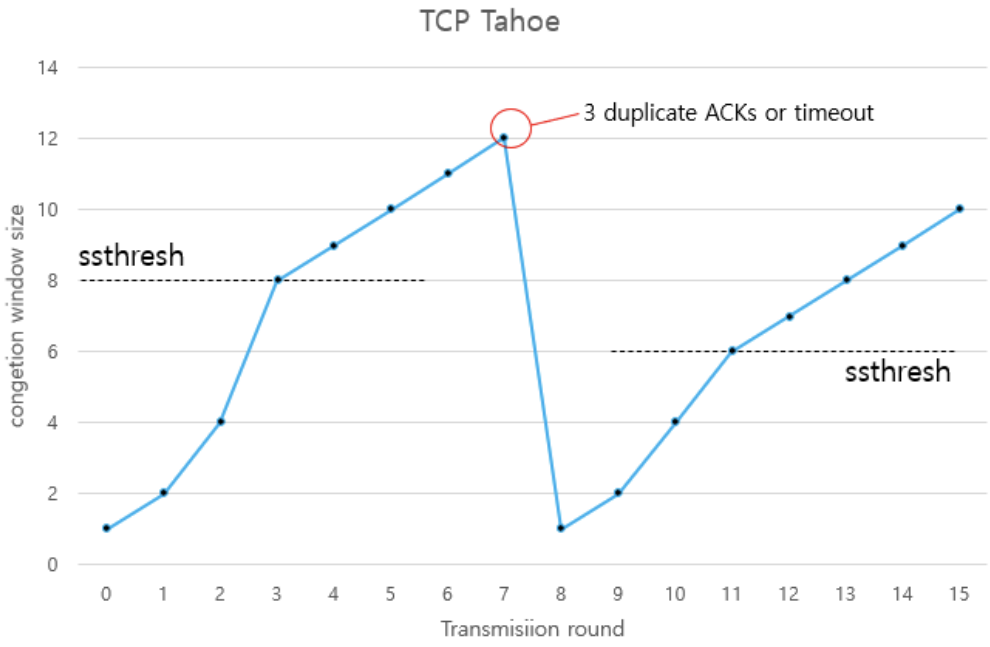
AIMD 방식은 윈도우 크기를 선형적으로 증가시키기 때문에, 제대로된 속도가 나오기까지 시간이 오래 걸린다. 반면 Slow Start는 윈도우 사이즈를 패킷 손실이 일어날 때까지 지수승으로 증가시킨다.



- 초기 윈도우 사이즈 : 1 MSS
- 매 RTT마다 윈도우 크기를 2배로 키운다. (ex: 1, 2, 4, 8, 16)
- 패킷 손실을 감지하면 window size를 1MSS로 줄인다.
- 처음에는 윈도우 사이즈가 1이라 속도가 느리나 지수승으로 윈도우 사이즈가 커지므로 속도가 빠르게 증가한다.
- 처음에는 네트워크의 수용량을 예상할 수 있는 정보가 없지만, 한번 혼잡 현상이 발생하고 나면 네트워크의 수용량을 어느 정도 예상할 수 있다.
- 그러므로 혼잡 현상이 발생하였던 window size의 절반까지는 이전처럼 지수 함수 꼴로 창 크기를 증가시키고 그 이후부터는 완만하게 1씩 증가시킨다.

TCP Tahoe

> slow start threshold(ssthresh)는 여기까지만 Slow Start를 사용하겠다는 임계점이다.

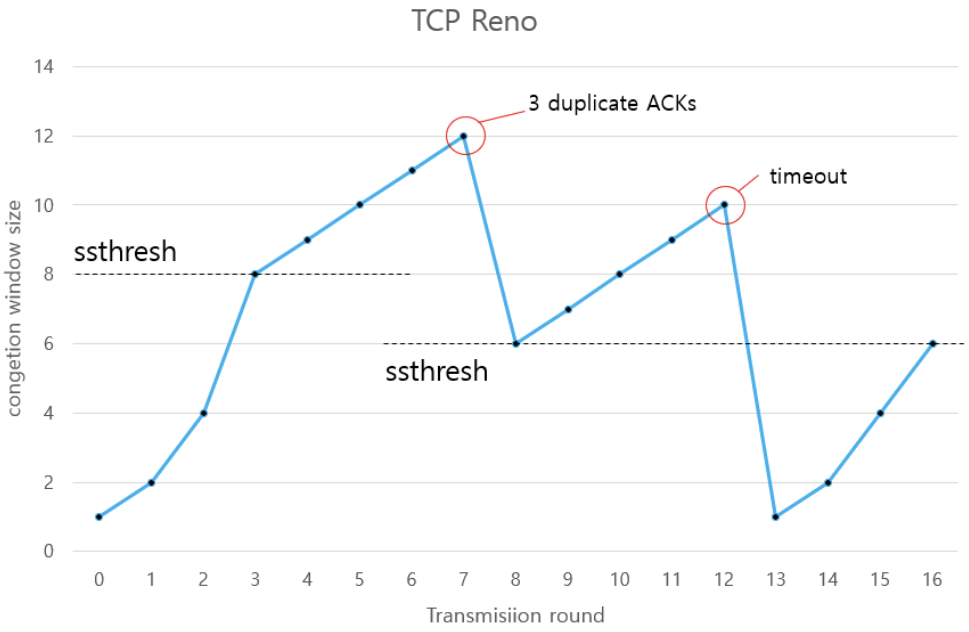


TCP Tahoe는 처음에는 Slow Start를 사용하다가 임계점에 도달하면 AIMD 방식을 사용한다.

- 처음 윈도우 사이즈는 1이다.
- 임계점까지는 Slow Start 방식을 사용한다. (window size 2배로 증가)
- 임계점부터는 AIMD 방식을 사용한다. (window size 1씩 증가)
- **3 duplicate ACKs 혹은 timeout을 만나면** 임계점을 **윈도우 사이즈의 절반으로 줄이고 window size를 1로 줄인다.**

TCP Tahoe 방식은 3 duplicate ACKs를 만나고 window size가 다시 1부터 키워나가야 하므로 속도가 느리다. 이를 해결할 수 있는 방식이 TCP Reno이다.

TCP Reno



TCP Reno는 TCP Tahoe와 비슷하지만 **3 dupicate ACKs와 timeout을 구분**한다는 점이 다르다.

- 처음 window size는 1 MSS이다.
- 임계점까지는 Slow Start를 사용한다(window size가 2배씩 증가한다)
- 임계점부터는 AIMD 방식을 사용한다(window size가 1씩 증가한다)
- 3 duplicate ACKs를 만나면 window size를 절반으로 줄이고 임계점을 그 값으로 설정한다.
- timeout을 만나면 window size를 1로 줄인다. 임계점은 변하지 않는다.