TCP 흐름 제어, 오류 제어, 혼합 제어

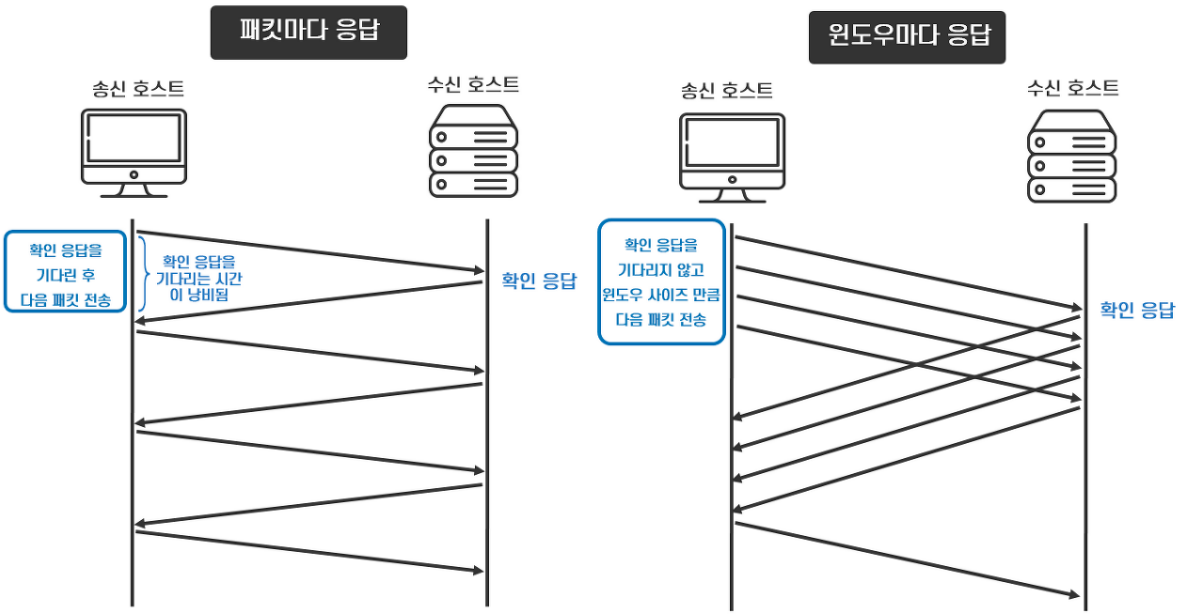
TCP/IP

- **TCP(전송 제어 프로토콜)과 IP(인터넷 프로토콜)을 아울러 지칭하는 용어**를 의미한다.
  - **IP** → 패킷을 **정확한 목적지로 빠르게 보내는 데 집중**한다.
  - **TCP** → 패킷에 **문제가 없는지 점검**하고, **정확한 애플리케이션으로 이동**하도록 유도한다.
  - **두 방식을 조합해 인터넷 데이터 통신을 하는 것을 TCP/IP 라고 한다.**

TCP 흐름 제어

- 통신 신뢰성 확보를 위해 하나의 **세그먼트**를 전송할 때마다 확인 응답을 거치면 전송자 입장에서는 응답이 돌아올 때까지 아무 일도 하지 않아 **전송 속도가 지연**된다.
  - 이 방식을 `Stop and Wait` 이라고 부르며, **간단하지만 속도 자체가 너무 느리다.**

    → **통신 속도를 높이기 위해 TCP는 `Sliding Window` 흐름 제어 방식을 사용한다.**



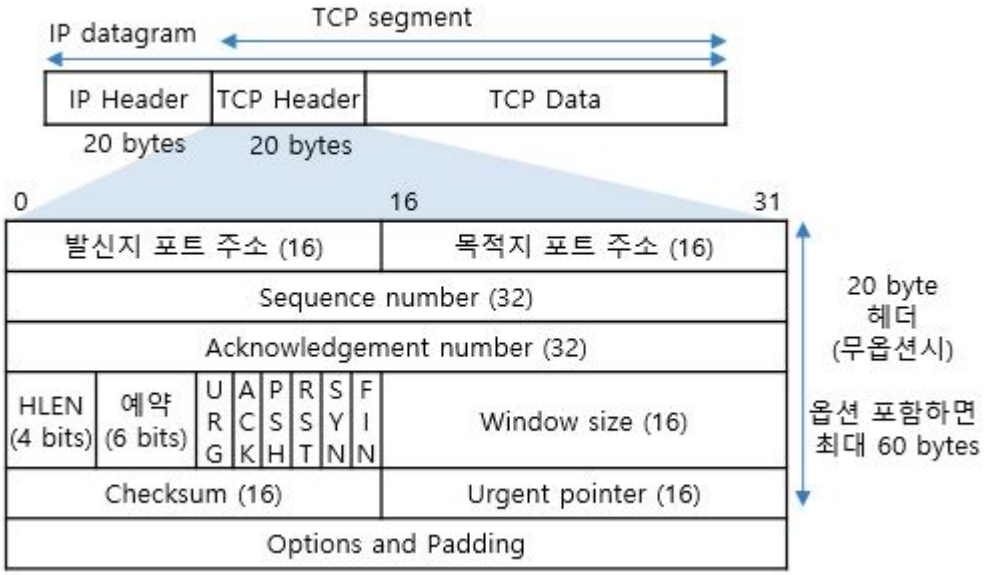
- 매번 확인 응답을 기다리는 대신, **세그먼트를 연속해서 보내고 난 후 확인 응답을 반환**함으로서 **통신 속도의 효율을 높인다.**

- 하지만 수신자에게 세그먼트가 계속해서 쌓일 수 있는데, 이 문제는 어떻게 해결할까?
  - **받은 세그먼트를 일시적으로 보관하는 장소인 버퍼**를 이용해 해결한다.

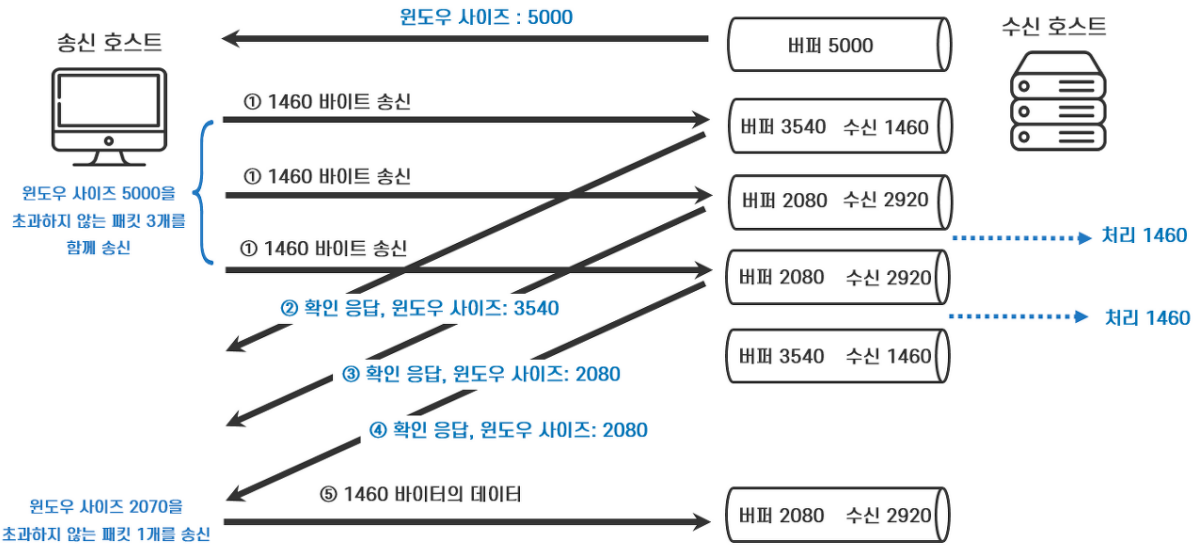
> **버퍼**는 컴퓨터 메모리 상에 위치해 송수신하는 패킷을 일시적으로 저장하는 장소를 의미한다.

- 그러나, 전송자가 버퍼를 통해 많은 데이터를 한 번에 보내면 수신자 입장에서 제 때 처리하지 못할 수 있다.
  - 전송자의 데이터 전송 속도보다 수신자의 데이터 수신 속도가 떨어지면 **처리할 수 없는 데이터가 흘러 넘치는 오버플로우 문제가 발생**한다.

    → **오버플로우가 발생하지 않도록 버퍼의 한계 크기를 알고 있어야 한다.**



- **버퍼의 한계 크기**는 **TCP 헤더의 윈도우 크기 값**에 해당한다.
  - 윈도우 크기는 **얼마나 많은 용량의 데이터를 저장할 수 있는지**를 나타낸다.
  - 즉, **데이터를 전송할 때 한 번에 전송할 수 있는 전체 세그먼트 크기를 나타내는 것**이다.
  - **전송자는 윈도우 크기만큼 수신자의 확인 응답을 기다리지 않고 세그먼트를 전송할 수 있다.**



- 윈도우 크기는 `3-Way Handshake` 과정에 의한 연결 확립 과정에서 **수신자가 초기값을 결정**한다.
    - 수신자는 확인 응답을 보낼 때 TCP 헤더에 윈도우 크기를 지정하고, **현 상태에서 어느 정도까지 수신이 가능한지 수시로 알려준다.**

- 이렇게, **TCP는 수신자가 윈도우 크기를 변경하는 방법으로 전송자에게 데이터 전송량을 지시함으로써 데이터 흐름을 제어하고 전송 효율을 높인다.**

TCP 오류 제어

- TCP는 기본적으로 `ARQ(Automatic Repeat Request)`, **재전송 기반 오류 제어를 사용**한다.
    - **통신 중 오류 발생 시 전송자가 수신자에게 해당 데이터를 다시 재전송**한다.
    - **그러나, 재전송 작업 자체가 일을 반복하는 것이므로 비효율적이며, 재전송 과정을 최대한 줄여야 한다.**

오류 발생 파악 방법

1. **수신자가 전송자에게 명시적으로 `NACK`을 보낸다.**
2. **전송자에게 `ACK`가 오지 않거나 중복된 `ACK`가 계속 오면 오류가 발생했다고 추정한다.**

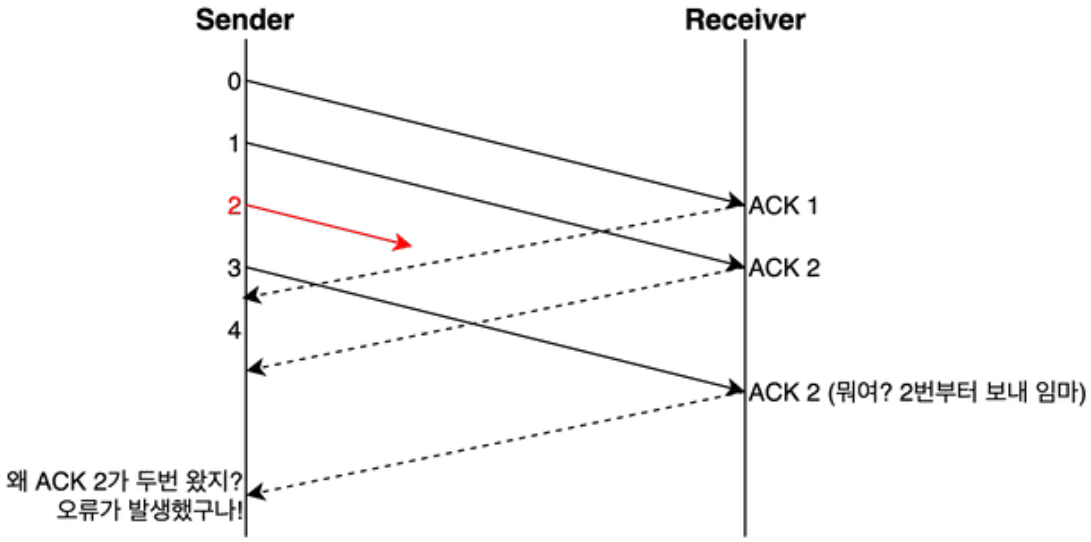
- `NACK` 을 사용하는 방법은 명확하고 구현이 쉬워보이지만, 수신자가 전송자에게 `ACK` 를 보낼지, `NACK` 을 보낼지 **선택해야 하는 로직이 추가적으로 강요**된다.

  → 그래서, **일반적으로는 `ACK` 만을 사용해 오류를 추정하는 2번 방식이 주로 사용**된다.

- 전송자에게 `ACK`가 오지 않는 **타임아웃 발생 조건**은 아래 두 가지와 같다.
    1. **전송자가 보낸 데이터가 중간에 유실돼 수신자가 아예 데이터를 받지 못하며 `ACK` 를 보내지 못했을 때**
    2. **수신자가 제대로 응답했지만 해당 `ACK` 패킷이 유실됐을 때**

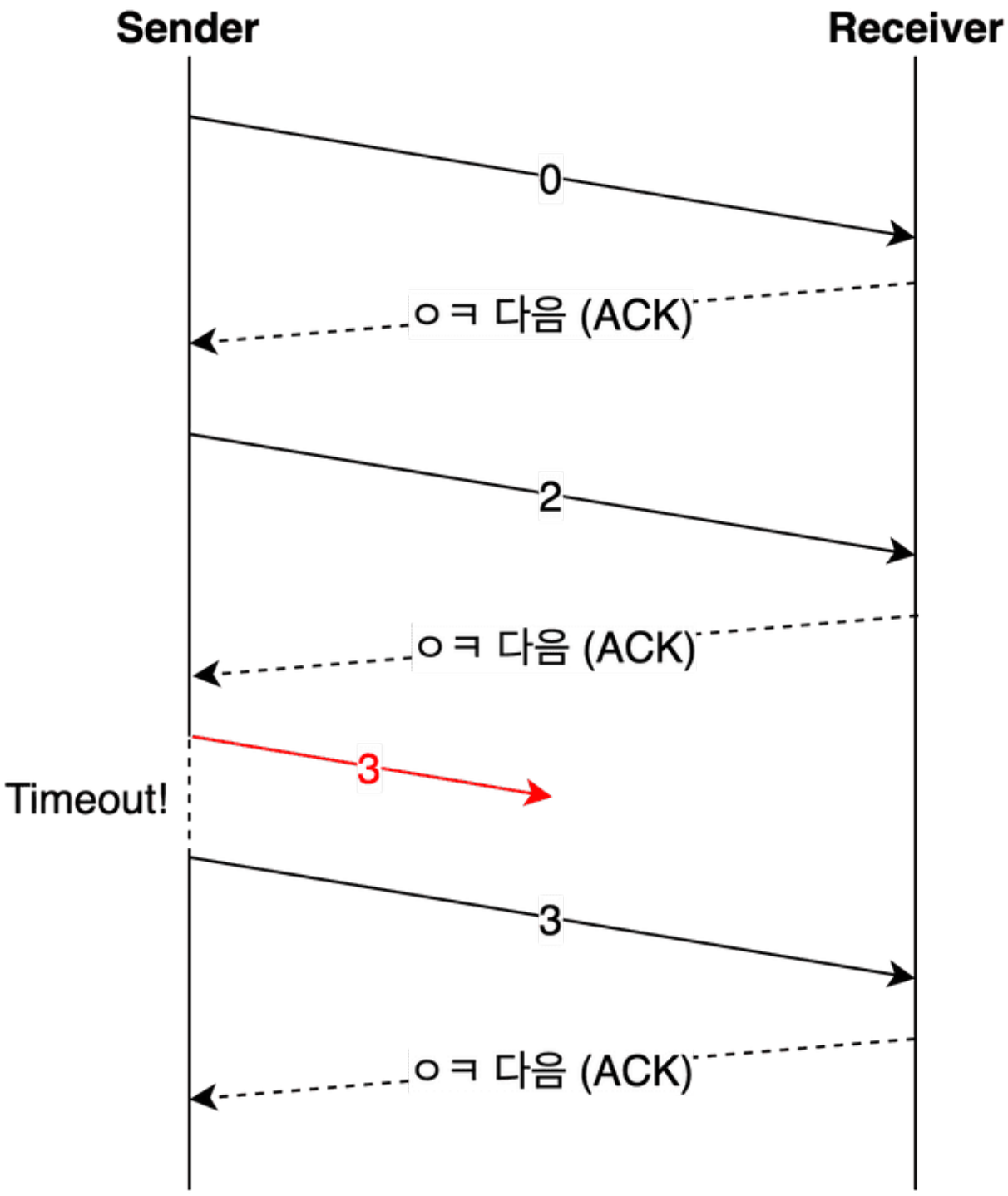
→ 즉, **전송자가 데이터를 전송했지만 수신자가 응답하지 않고 일정 시간이 경과한 경우**이다.

- **전송자가 중복된 `ACK`가 왔을 때 오류라고 판별하는 방법**은 아래 이미지와 같다.



  - **수신자가 계속해서 2번 데이터를 보내달라고 요청하는 것으로, 이 때 전송자는 자신이 보낸 2번 데이터에 문제가 발생했음을 알 수 있다.**

Stop And Wait

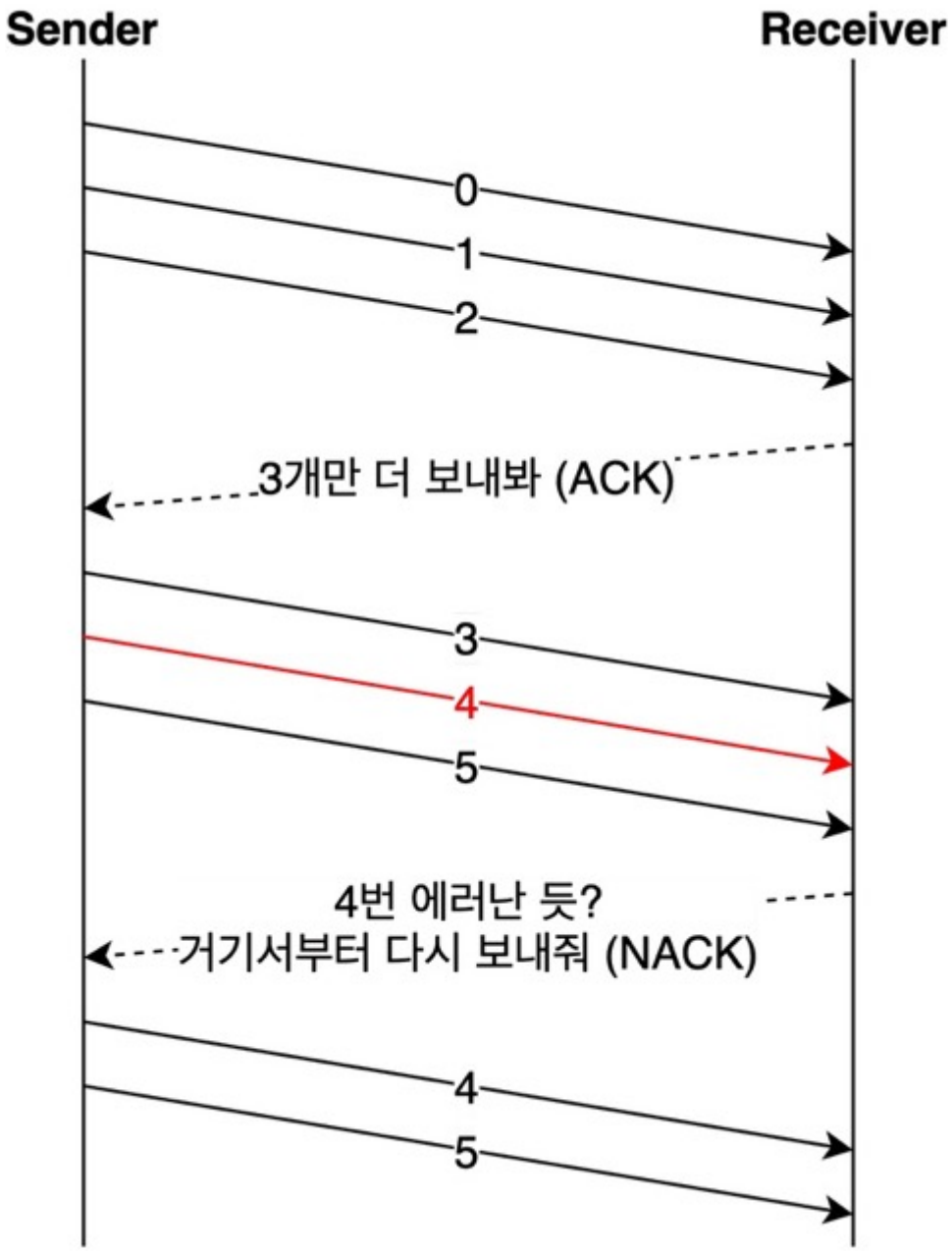


- 흐름 제어에서 한 번 나왔던 방법으로, **한 번 데이터를 보냈을 때 `ACK` 응답이 올 때까지 대기하다가 다음 데이터를 보내는 방식**이다.
    - 제대로 받았다는 응답이 오지 않으면 계속해서 데이터를 재전송하기 때문에, 흐름 제어와 오류 제어 모두 가능하다.

- 그러나 `Sliding Window` 를 사용해 **흐름 제어**를 하는 경우에서 `Stop And Wait` 을 사용해 **오류 제어**를 해버리면 **데이터를 연속적으로 보내는** `Sliding Window` **의 장점을 잃어버리게 된다.**

Go Back N

- **데이터를 연속적**으로 보내다가 **오류 발생 시 어느 데이터부터 오류가 발생했는지 검사하는 방식**이다.



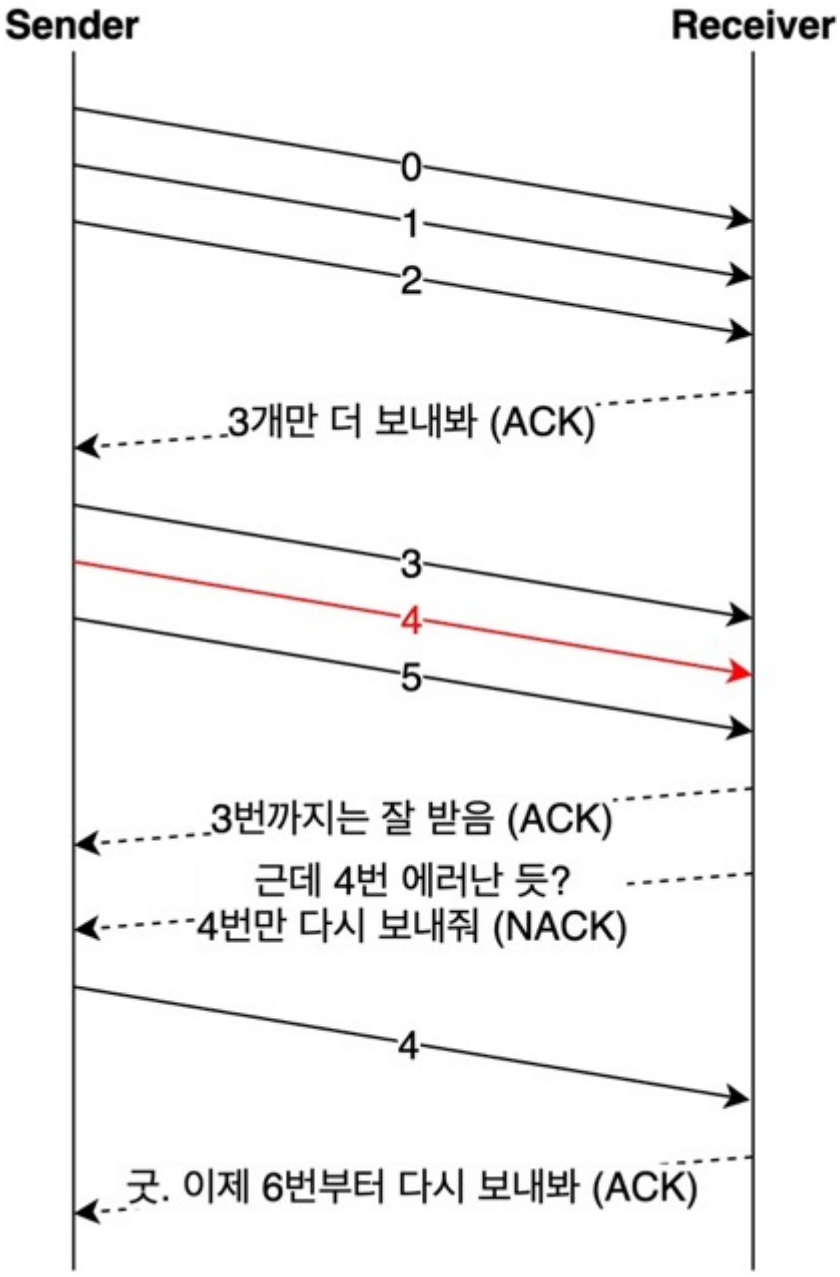
- `Go Back N` 방식을 사용하면 **데이터를 연속적으로 보낸 후 한 개의** `ACK` **나** `NACK` **만을 사용해 수신자 측의 처리 상황을 파악**할 수 있다.
  - 즉, `Sliding Window` **방식과 함께 사용하면 효율이 올라간다.**

- 위 그림에서 **수신자가 4번 데이터부터 오류 발생을 감지하고 전송자에게 4번부터 재전송을 요청**한다.
  - 즉, **4번 데이터 이후 수신자는 받았던 데이터를 모두 폐기**하고 전송자에게 `NACK` 을 보낸다.

- **전송자는 `NACK` 을 받으면 4번 데이터와 이후 전송 데이터를 모두 다시 전송해야 한다.**
- 즉, **오류 발생 데이터로 되돌아가 다시 전송**해야 하기 때문에 `Go Back N` 방식인 것이다.

Selective Repeat

- **선택적 재전송**이라는 뜻으로, **에러가 발생한 데이터만 재전송을 요청하는 방식**이다.
  - **즉, `Go Back N` 방식을 개선한 것이다.**



- 굉장히 효율적인 것처럼 보이지만, **수신자 측 버퍼에 쌓인 데이터가 연속적이지 않다는 치명적 단점**이 있다.
  - 위 그림에서도 수신자 측 버퍼에 중간에 폐기 처분된 4를 제외한 `0, 1, 2, 3, 5` 만 버퍼에 존재할 것이다.
  - **만약 전송자가 4를 재전송하면 수신자는 4를 버퍼 중간 어딘가에 끼워넣어 데이터를 정렬해야 한다.**
  - 이 때, 같은 버퍼 안에 데이터를 정렬할 수 없기 때문에 별도 버퍼가 필요하다.

- **결국 재전송 과정이 빠진 대신 재정렬이라는 과정이 추가된 것이다.**
  - **재전송**이 좀 더 이득인 상황 → `Go Back N`
  - **재정렬**이 좀 더 이득인 상황 → `Selective Repeat`

- **복잡한 네트워크에 다시 접근하는 것보다 수신자 측이 재정렬을 하는 것이 이득인 경우가 많아 기본적으로** `Selective Repeat` **방식을 사용**한다.

TCP 혼잡 제어

- 어떤 이유로 전송이 느려지는지는 파악하기 힘들지만, **단순히 느려지고 있다**라는 과정은 **각 호스트에서도 충분히 파악**할 수 있다.
  - 만약 **전송이 느려졌을 때 흐름 제어 기법만 사용하면 재전송 작업이 반복될 수 밖에 없다.**

    → **혼잡 제어는 현재 네트워크의 혼잡 상태를 파악**하고, **혼잡 상태를 해결하기 위해 전송자의 데이터 전송을 제어하는 것**을 의미한다.

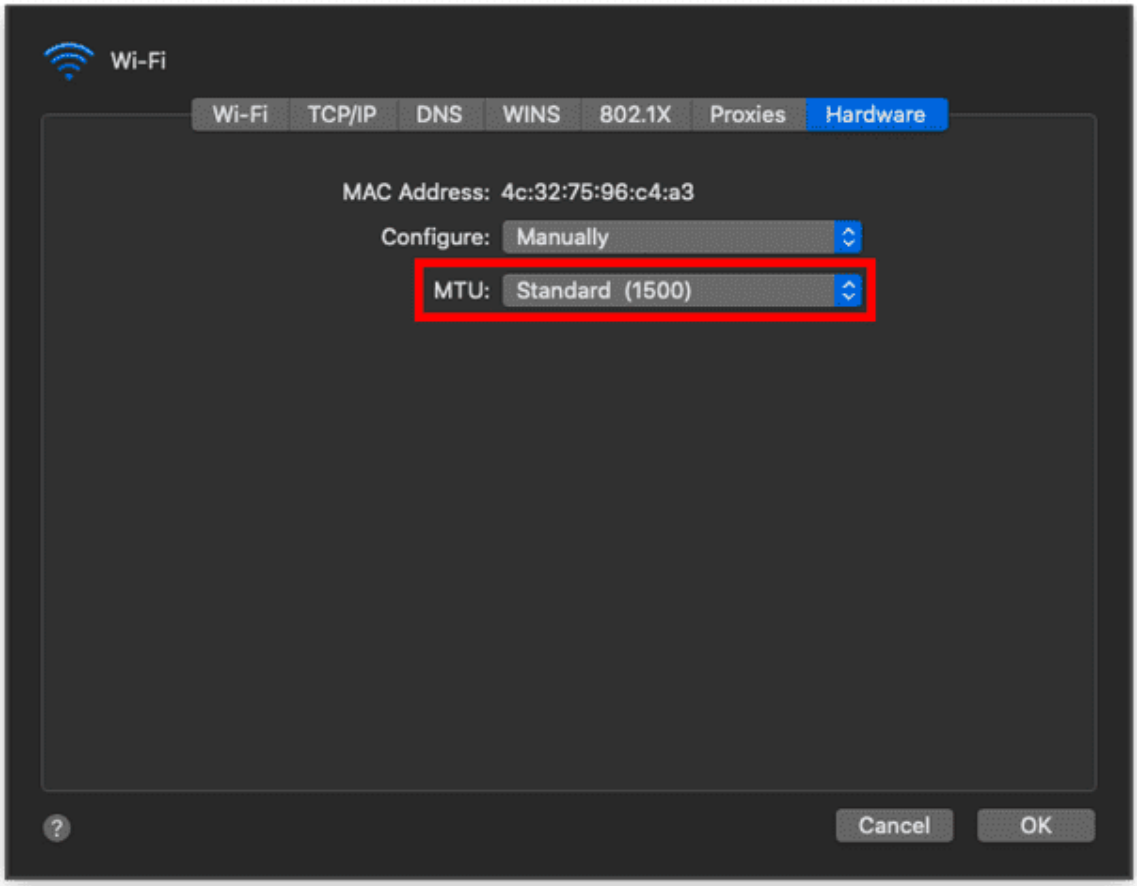
혼잡 윈도우(Congestion Window, CWND)

- 전송자는 자신의 최종 윈도우 크기를 정할 때 **수신자 측에서 보내준 윈도우 크기인 수신자 윈도우(RWND)**, **전송자 본인이 직접 네트워크의 상황을 고려해 정한 윈도우 크기인 혼잡 윈도우(CWND)** 중 **더 작은 값을 사용**한다.

- 통신 도중에는 ACK가 유실된다거나, 타임아웃이 난다는 등의 정보로 혼잡 상황을 유추할 수 있지만, **통신 시작 전에는 정보가 없어 혼잡 윈도우의 크기를 초기화하기 애매**하다.

  → **이 때 MSS(Maximum Segement Size)를 사용한다.**

  - `MSS = MTU - (IP 헤더 길이 + IP 옵션 길이) - (TCP 헤더 길이 + TCP 옵션 길이)`
  - `MTU(Maximum Transmission Unit)`**는** 한 번 통신 때 보낼 수 있는 최대 단위를 의미한다.

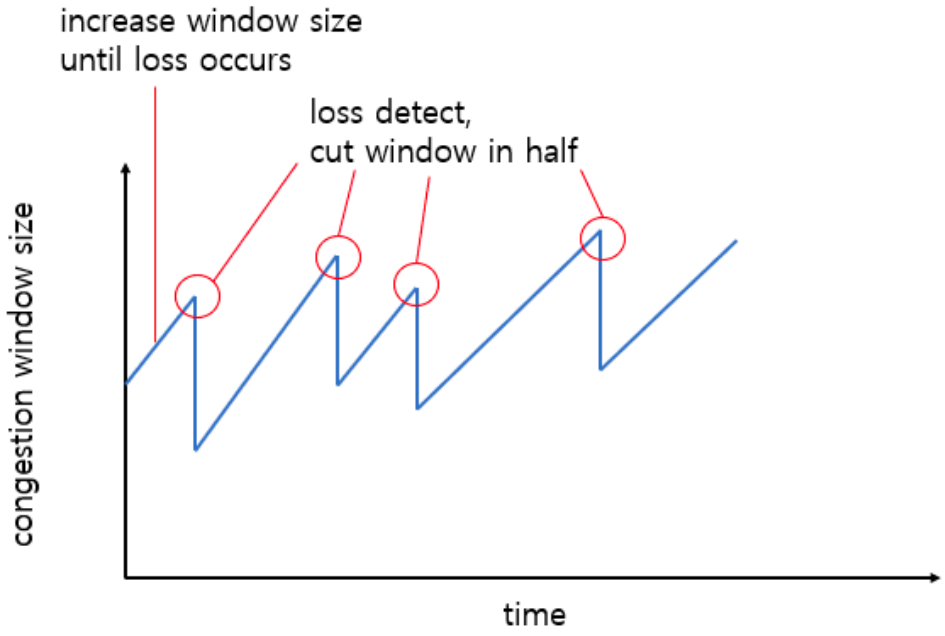


- 즉, `MSS`는 IP 헤더, TCP 헤더 등 **데이터가 아닌 부분을 전부 자르고 진짜 데이터를 담을 수 있는 공간**이 얼마나 남았는지를 나타낸다.
  - OSX는 `MTU` 기본 값으로 1500 바이트가 설정되어 있으며, TCP와 IP 헤더 크기가 각각 20 바이트라고 한다면 `MSS`는 $1500 - 40 = 1460$ 바이트가 될 것이다.

- 전송자는 처음 통신을 시작할 때 계산한 `MSS`를 사용, **혼잡 윈도우 크기를 `1 MSS`로 설정**한다.
  - 이 후 **통신을 진행하며 네트워크 혼잡 상황을 고려해 혼잡 윈도우 크기를 증가 혹은 감소**시킨다.

혼잡 회피 방법

AIMD (Additive Increase Multicative Decrease)

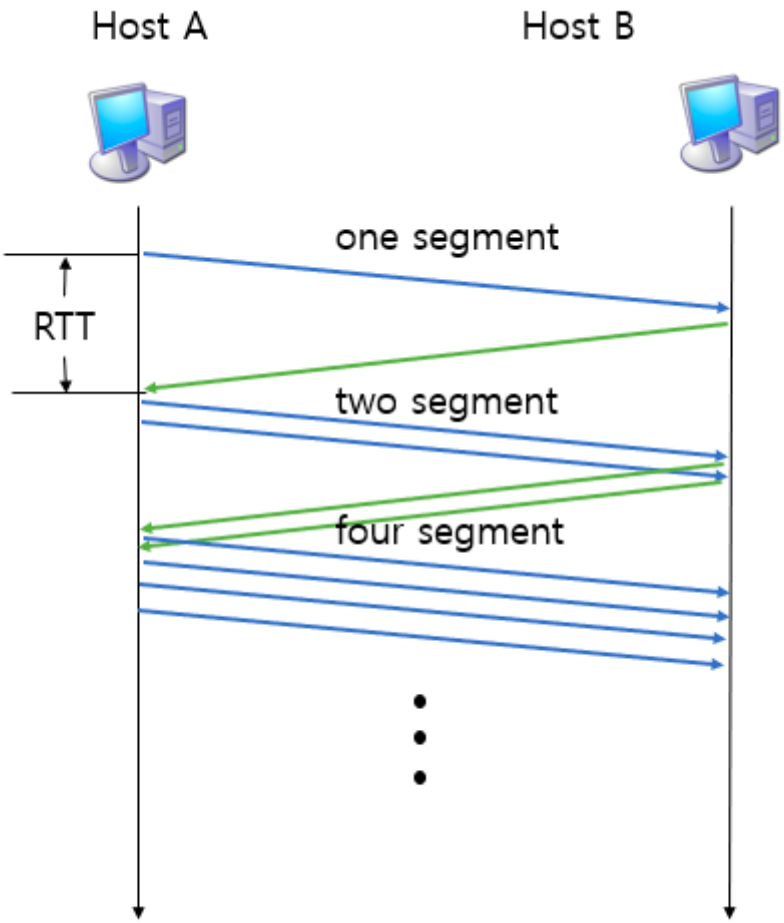


- **우리 말로 합 증가 / 곱 감소 방식을 의미한다.**
- **네트워크에 문제가 없어 전송 속도를 빠르게 하고 싶다면 윈도우 크기를 `1` 씩 증가**시킨다.
    - 그러나, 중간에 데이터가 유실되거나 응답이 오지 않는 **혼잡 상태가 감지**되면 **윈도우 크기를 반으로 줄인다.**

- **간단하지만 공평한 방식**으로, 시간이 가면 갈수록 **네트워크에 참여한 순서와 관계없이 모든 호스트들의 윈도우 크기가 평행 상태로 수렴**하게 된다.
    - 네트워크가 혼잡해지면 혼잡 윈도우 크기가 작은 호스트보다 **혼잡 윈도우 크기가 큰 호스트가 데이터를 무리하게 보내려다가 유실될 확률이 크다.**
    - 그렇기 때문에, **혼잡 윈도우가 큰 호스트는 혼잡 윈도우 쿠기를 줄여 혼잡 상황을 해결**한다.
    - 이 때, **남은 대역폭을 활용해 나중에 들어온 호스트는 자신의 혼잡 윈도우 크기를 키울 수 있다.**
    - **결론적으로 평행 상태로 수렴하게 된다.**

- 그러나, 네트워크 대역이 남는 상황에서도 윈도우 크기를 너무 조금씩 늘리며 접근하기 때문에 **네트워크의 모든 대역을 활용해 제대로 된 속도로 통신하기까지 시간이 오래 소요**된다.

Slow Start

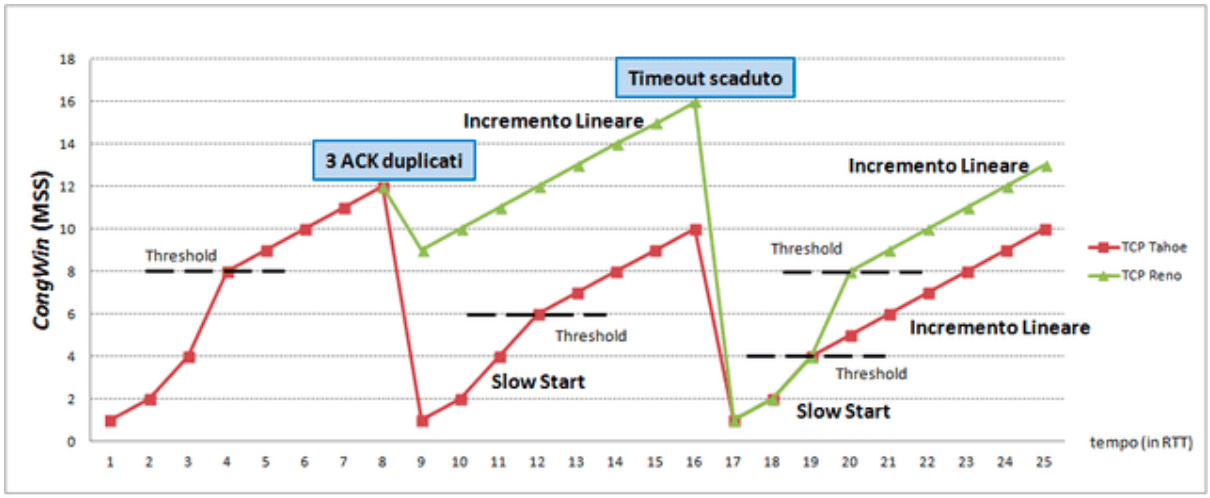


- 기본적인 원리는 AIMD와 비슷하지만, **윈도우 크기를 증가시킬 때 지수적으로 증가**시킨다.
  - 그러다가, **혼잡이 감지되면 윈도우 크기를 1로 줄여버린다.**

- 처음에는 **윈도우 크기가 조금 느리게 증가할지 몰라도, 시간이 갈수록 윈도우 크기가 점점 빠르게 증가**한다.
  - 즉, **AIMD보다 윈도우 크기를 훨씬 더 빠르게 키울 수 있다.**

혼잡 제어 정책

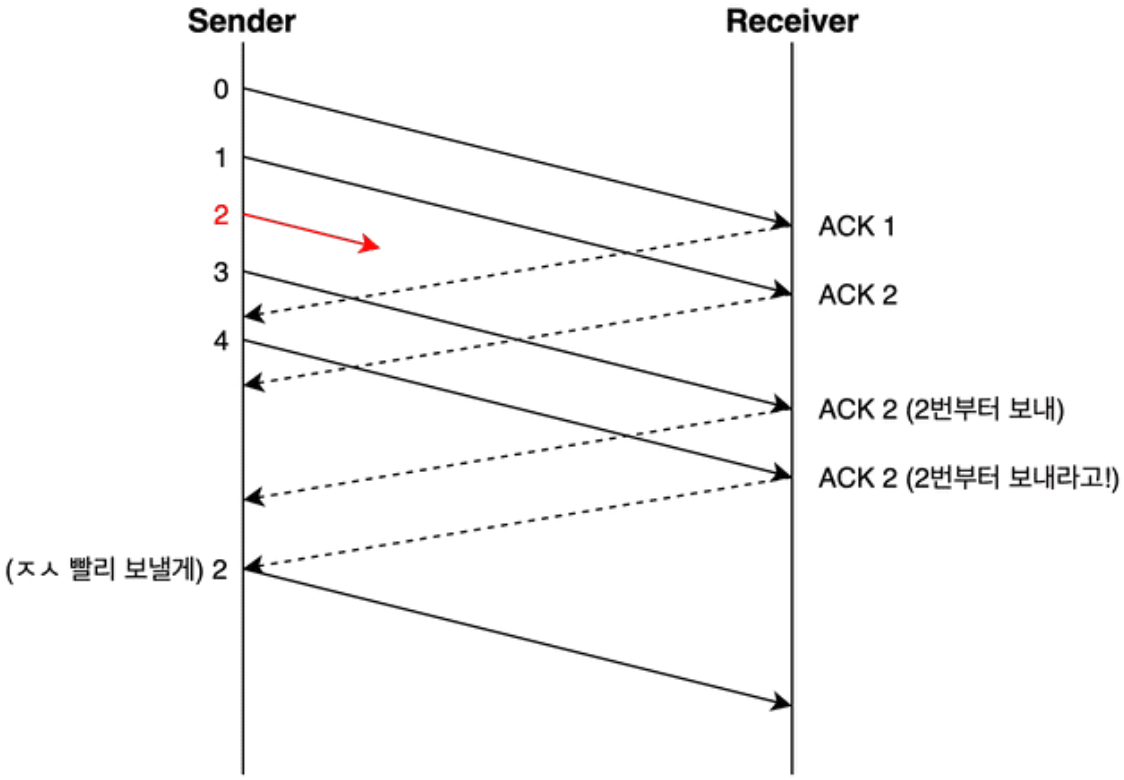
- **혼잡 제어 정책**은 **공통적으로 혼잡이 발생하면 윈도우 크기를 줄이거나, 혹은 증가시키지 않으며 혼잡을 회피하는 전제**를 깔고 있다.
  - 가장 대표적이고 유명한 정책으로 `Tahoe`, `Reno` 가 있다.



붉은 색이 Tahoe, 녹색이 Reno 의 윈도우 크기

- `Tahoe`, `Reno` **모두 처음에는** `Slow Start` **방식을 사용**하다가 **네트워크가 혼잡하다고 느꼈을 때** `AIMD` **방식으로 전환**하는 방법을 사용한다.
  - `3 ACK Duplicated`, `Timeout` 에 관한 시나리오가 발생하면 윈도우 크기를 줄인다.

- 간단하게 위 그래프에 쓰여진 용어에 대한 설명은 아래와 같다.
- **3 ACK Duplicated**
  - **전송자 측이 3번 이상 중복된 승인 번호를 받은 상황**으로, 어떤 이유로 인해 **수신자 측이 특정 일련 번호 이후의 데이터를 제대로 처리하지 못한 상황**을 의미한다.
  - 단, 패킷 전송 방식을 사용하는 TCP의 특성 상 수신자 측이 받는 패킷 순서가 늘 순서대로 받는다는 보장은 없으므로, **한 두개의 중복 승인 번호가 발생한 것으로 네트워크가 혼잡하다고 판단하지 않는다.**

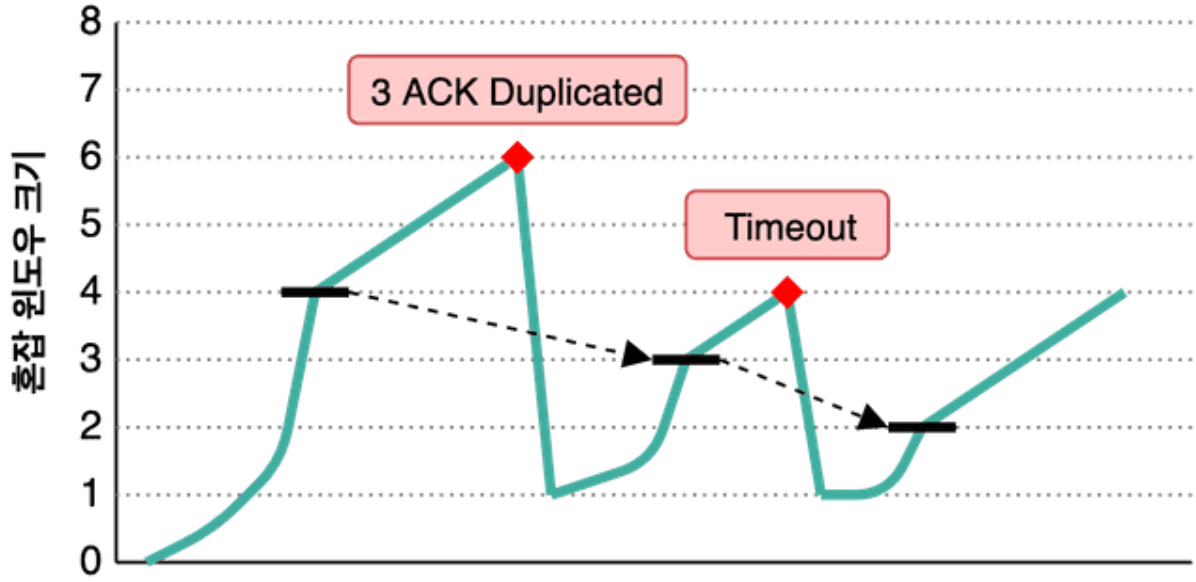


- **3번의 중복 승인 번호**로 인해 **전송자 측이 해당 승인 번호에 해당하는 데이터를 전송**하면, 수신자 측은 `Go back N` 이나 `Selective Repeat` 과 같은 오류 제어 방식에 따라 **다음에 어떤 패킷부터 보내주어야 하는지 알린다.**
- 이런 상황일 때, **전송자 측은 자신이 설정한 타임아웃 시간이 지나지 않아도 해당 패킷을 바로 재전송**할 수 있으며, 이 기법을 **빠른 재전송(Fast Transmit)** 이라고 한다.
  - 빠른 재전송 기법을 사용하지 않는다면 **전송자 측은 타임아웃 시간이 지난 후에야 대처가 가능**하기에 에러 데이터를 재전송할 때까지 **시간이 낭비**되게 될 것이다.

- **Slow Start 임계점(ssthresh)**
  - **해당 지점까지만** `Slow Start` **를 사용하겠다는 의미**를 가진다.
  - **값을 사용하는 이유**로는 **계속해서** `Slow Start` **를 사용**하면 **어느 순간부터 윈도우 크기가 기하급수적으로 늘어나 제어가 힘들어지고**, **혼잡이 예상되는 상황에서는 조금씩 증가시키는 편이 안전**하기 때문이다.
  - 특정 임계점(Threshold)을 설정해놓고 임계점을 넘기면 `AIMD` 방식을 사용해 선형적으로 윈도우 크기를 증가시킨다.

TCP Tahoe

- `Slow Start` 를 사용한 **혼잡 제어 정책의 초기 버전**이다.
  - **위에서 설명한 빠른 재전송 기법이 처음으로 도입**되었다.

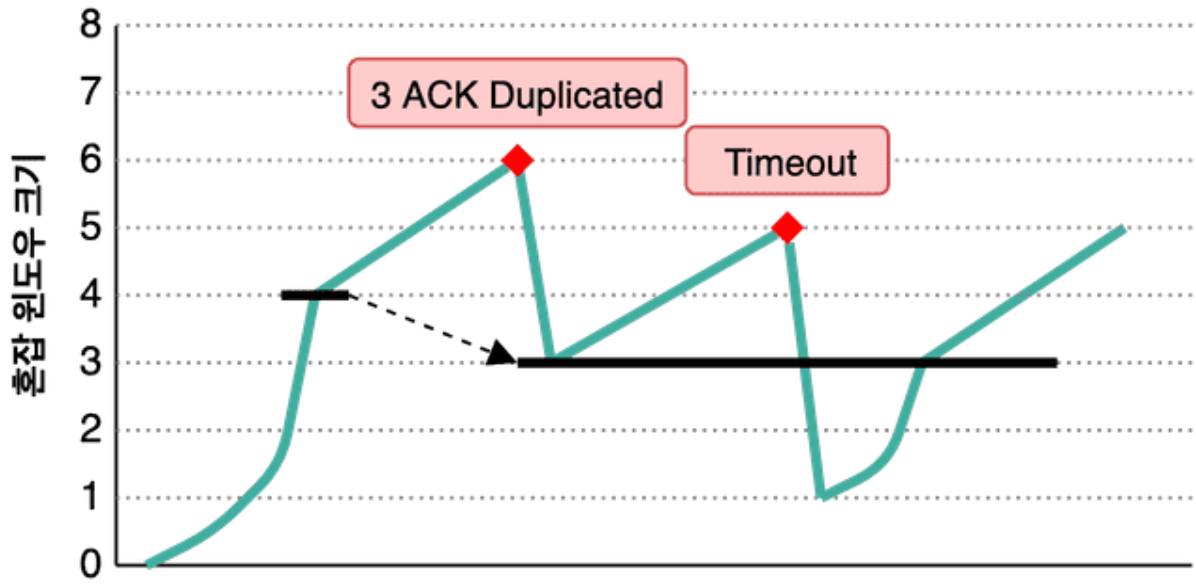


- 초기엔 `Slow Start` 를 사용해 **윈도우 크기를 빠르게 증가**시킨다.
  - `ssthresh` **를 만난 이후부터는** `AIMD` **방식을 사용해 선형적으로 윈도우 크기를 증가시킨다.**
  - 그러다가 `ACK Duplicated` 나 `Timeout` 이 발생하면 **혼잡이 발생했다고 판단**하고, `ssthresh` 와 **자신의 윈도우 크기를 수정한다**.

- 초반 `Slow Start` 구간에서 **윈도우 크기를 키울 때 너무 오래걸린다는 단점**이 있다.
  - 전체적으로 보면 AIMD 방식 보다는 빠르지만, **혼잡 상황이 발생했을 때 다시 윈도우 크기를 1부터 키워나가야 한다는 점은 낭비를 유발**한다.

TCP Reno

- `Tahoe` **와 마찬가지로** `Slow Start` **로 시작해 임계점을 넘어서면** `AIMD` **방식으로 변경**한다.
- 그러나 **명확한 차이점으로** `3 ACK Duplicated` **와** `Timeout` **을 구분한다는 점**이 있다.
  - **중복** `ACK` **가 발생했을 때 윈도우 크기를 1로 줄이지 않고, AIMD처럼 반으로 줄인다.**
  - **추가로** `sshthresh` **를 줄어든 윈도우 값으로 수정**한다.

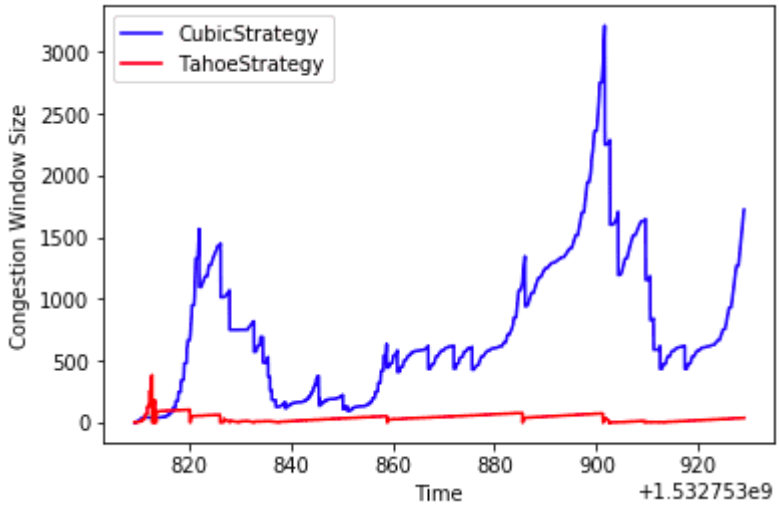


- `Tahoe` 와 달리 `3 ACK Duplicated` 발생 시 **혼잡 윈도우 크기를 반으로 줄인 후** `AIMD` **방식을 적용**한다.
  - 이 방식은 기존 `Tahoe` 에 비해 **빠르게 기존 윈도우 크기로 도달할 수 있어 빠른 회복(Fast Recovery)**이라고 불린다.

- 그러나 `Timeout` 에 의해 **데이터가 손실**되면 `Tahoe` 와 마찬가지로 **윈도우 크기를 1로 줄인 후** `Slow Start` **방식을 진행**한다.
  - **이 때,** `ssthresh` **는 변경하지 않는다.**

- 정리하면 `Reno` 방식은 **중복 상황과 타임아웃이 발생한 상황을 구분하며 대처를 다르게 하는 것이다.**
    - **어느 정도 혼잡 상황에 경중을 따지고 있다는 것을 의미한다.**

최근의 방식



최근의 혼잡 제어 정책인 Cuibc, 이전의 혼잡 제어 정책 Tahoe의 비교

- 최근에는 **대역폭의 크기가 상당히 많이 증가**했기 때문에 `Tahoe`, `Reno` 를 적용하기엔 부적절하다.
    - **전송자 측이 자신의 혼잡 윈도우 크기를 크게 늘려도 문제가 발생할 확률이 낮아졌음**을 의미한다.

      → **최근 혼잡 제어 정책은 얼마나 더 빠르게 혼잡 윈도우를 키우고, 어떻게 혼잡 감지를 똑똑하게 할 것이냐에 대해 초점이 맞춰져있다.**

References

- 모두의 네트워크
- https://velog.io/@haero_kim/TCP-흐름제어-기법-살펴보기
- https://better-together.tistory.com/140?category=887984

- https://evan-moon.github.io/2019/11/26/tcp-congestion-control/