Network Manias
Analyze Trends, Technologies and Market

Netmanias-wp-sub-112

TCP Loss Recovery Mechanism

September 8, 2003

연세대학교 전기전자공학과
고성능 멀티미디어 네트워크 연구실
김범준 박사 (bjkim@nasla.yonsei.ac.kr)

1. 서론

TCP의 혼잡 윈도우의 크기는 패킷 손실이 발생하지 않는 한 계속 증가한다. 반대로 말하면, 언젠가는 혼잡으로 인한 패킷 손실이 발생하게 된다. TCP 혼잡 제어 메커니즘에는 손실된 패킷을 복구하고 혼잡 윈도우의 크기를 재조정 하기 위한 기능이 포함되어 있는데, 이를 흔히 TCP 손실 복구 메커니즘이라고 한다. TCP 손실 복구 메커니즘은 패킷 손실을 감지했을 때 우선 이를 재전송에 의해서 복구하기 위한 시도를 하게 된다. 만약, 재전송에 의한 복구가 불가능한 경우 재전송 타임 아웃 이후에 패킷 전송이 다시 시작된다. 따라서, 손실된 패킷을 어느 정도 복구할 수 있는지의 여부는 전반적인 TCP의 성능에 큰 영향을 미친다. TCP의 손실 복구 메커니즘은 두 개의 기본적인 알고리듬인 **Fast Retransmit**과 **Fast Recovery**를 통해 동작하는데, 특히 Fast Recovery는 손실 복구 메커니즘의 성능 향상을 위해서 수 차례에 걸쳐서 수정되어 왔고, 이에 따라 흔히 TCP Tahoe, TCP Reno, TCP NewReno로 구분된다. 이 글에서는 이 세 가지 종류의 TCP 손실 복구 메커니즘의 동작을 비교하고, 추가적으로 Selective Acknowledgement option을 사용하는 TCP의 손실 복구 메커니즘의 동작에 대해서도 알아본다.

2. Fast Retransmit [1]

Fast Retransmit은 1988년 Van Jacobson에 의해서 제안되고 이를 구현한 TCP를 흔히 TCP Tahoe라고 한다. 모든 TCP의 손실 복구 과정은 Fast Retransmit으로 시작된다. Fast Retransmit의 동작을 설명하기에 앞서, 중복 승인 (Duplicate Acknowledgement)을 이해하는 것이 필요하다. 이를 위해 다음 그림 1에 나타난 윈도우의 크기가 4인 상태에서 패킷 1~4까지의 패킷을 전송하고 이들 중 패킷 2가 손실된 경우를 생각해보자.

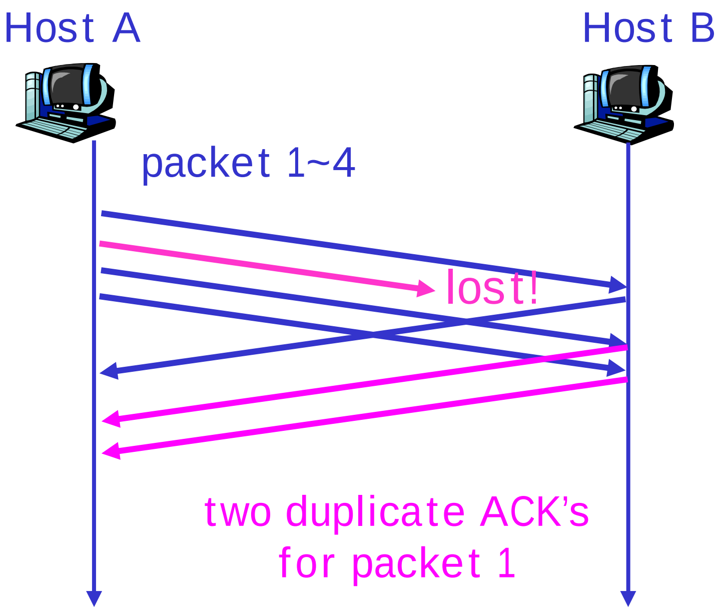

그림 1. 중복 승인 패킷의 예

TCP는 패킷을 수신할 때마다 지금까지 정상적으로 수신된 가장 나중의 패킷을 송신원에게 알려주는데 이를 흔히 누적 승인 방식 (Cumulative Acknowledgement)이라 한다. 패킷 1을 수신한 수신원은 송신원측에 패킷 2를 받을 차례라는 것, 즉 패킷 1까지는 정상적으로 수신되었다는 것을 알려주는 승인 패킷을 전송한다. 패킷 2는 전송 도중 손실되었으므로 수신원은 패킷 2를 수신하지 못한 상태에서 패킷 3을 수신한다. 따라서, 패킷 3을 수신한 수신원은 정상적으로 수신된 마지막 패킷은 패킷 1이고 패킷 2를 받을 차례라는 똑 같은 승인 패킷을 다시 전달하게 되는데 이를 흔히 중복 승인 (Duplicate Acknowledgement)라고 한다.

송신원이 세 개의 중복 승인을 받게 되면 해당 패킷은 손실되었다고 단정하고 이를 즉시 재전송하는데 이것이 Fast Retransmit 알고리듬의 기본 동작이다. Fast Retransmit 을 사용하지 않는 경우에는 패킷 손실이 발생하면 항상 재전송 타임 아웃이 만료된 후에 전송을 다시 시작하기 때문에, Fast Retransmit은 TCP의 성능을 크게 향상 시킨다. Fast Retransmit을 동작시키기 위해서 필요한 중복 승인 패킷의 수를 세 개로 정한 것에는 특별한 이론적인 배경은 없다. 다만, IP 패킷들은 서로 다른 경로로 전달되는 경우가 흔히 발생하기 때문에 TCP 패킷들 역시 순서가 바뀐 채로 전달될 수 있다. 이런 경우에 중복 승인을 한 개 받았을 때 무조건 그 패킷이 손실되었다고 가정하고 Fast Retransmit을 통해 재전송하게 되면 너무 많은 불필요한 재전송을 수행하게 되기 때문에 오히려 성능 저하의 요인이 될 수 있다. 반대로, 패킷 손실이 발생했는데도 이에 대한 세 개의 중복 승인을 수신할 수 없는 경우도 발생할 수 있는데 이 역시 TCP의 성능을 저하시키는 요인이 될 수 있다. 어쨌든 이에 대해서는 많은 연구가 진행되어 왔고 최근 제안된 Limited Transmit 알고리듬을 통해서 거의 100% 해결이 가능하다. (관심 있는 분들은 RFC 3042를 참조하시기 바랍니다.)

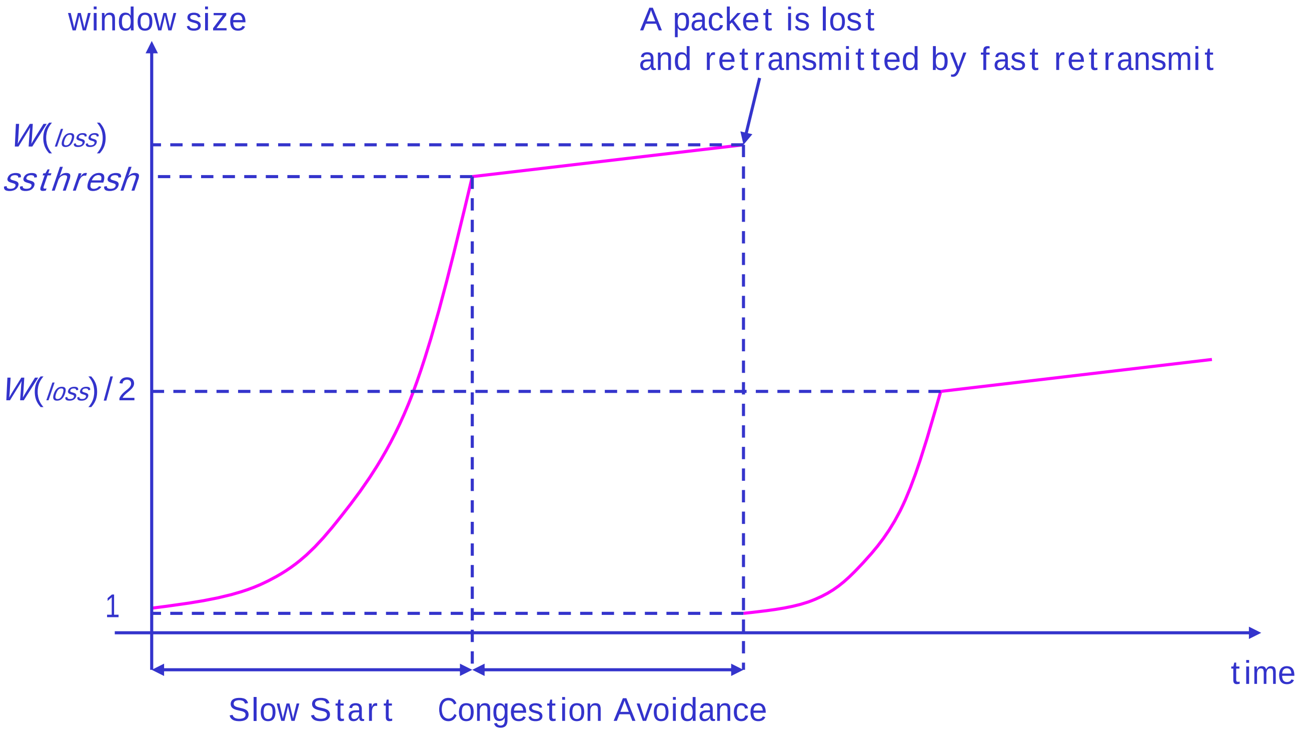

그림 2. Fast Retransmit에 의한 윈도우의 변화

그림 2는 Fast Retransmit에 따른 윈도우의 크기 변화를 보여준다. 윈도우의 크기가 W(loss)일 때 발생한 손실된 패킷을 Fast Retransmit에 의해 재전송한 이후 ssthresh의 값은 W(loss)/2로 설정된다. 이 값을 반으로 줄이는 것에는 특별한 이론적인 근거는 없다. Slow Start 상태인 송신원의 윈도우의 크기가 w일 때 w개의 패킷을 전송한 경우를 생각해보자. 이들에 의한 w개의 승인 패킷들을 수신할 때 윈도우의 크기는 이 보다 두 배인 2 w가 된고 2 w개의 패킷들이 전송된다. 이 시점에서 패킷 손실이 발생한다면 이를 Fast Retransmit에 의해 재전송한 후 ssthresh의 값을 현재 윈도우의 반인 w로 줄이게 되는데, 이는 패킷 손실이 발생하지 않은 대략적인 윈도우의 최대 크기가 w라는 것을 의미한다. 따라서, 이 후의 전송에서는 이 값까지는 윈도우를 Slow Start로 빨리 증가시키기 위한 설정이라고 할 수 있다. 또, 한가지는 TCP 뿐만 아니라 다른 윈도우 기반의 혼잡 제어 프로토콜에서 사용하는 윈도우의 부가적 증가 지수적 감소 (Additive Increase Multiplicative Decrease) 방식은 많은 송신원들이 존재할 때 이들의 공평성 (Fairness)을 제공하는데 있어 가장 성능이 좋다는 점을 들 수 있겠다. 이에 대해서는 많은 연구들을 통해 증명된 바 있다. 그림에서 볼 수 있듯, Fast Retransmit에 의해 패킷이 성공적으로 재전송된 후에는 다시 윈도우의 크기는 1로 설정되어 반으로 줄어든 ssthresh까지는 Slow Start 상태에서 증가한다. 만약, 손실된 패킷이 Fast Retransmit에 의한 복구가 불가능한 경우 재전송 타임 아웃이 만료된 후 다시 Slow Start가 시작되는데 이 때의 ssthresh의 값 역시 패킷 손실이 발생하기 전 윈도우의 반이다.

3. Fast Recovery [2]

초기 Fast Recovery 알고리듬은 90년 경에 처음 등장하는데, 이 알고리듬이 포함된 TCP 구현을 흔히 TCP Reno라고 한다. Fast Recovery 알고리듬의 등장은 당시 네트워크의 전송 속도의 급격한 증가와 밀접한 관련이 있다. 흔히, 송신원과 수신원 사이의 연결을 pipe로, 그리고 RTT와 대략적인 전송 속도의 곱을 Bandwidth-Delay-Product (BDP) 라고 하는데, 위성 네트워크와 같이 전송 속도와 전달 지연이 매우 큰 네트워크 상의 연결은 Long-Fat Pipe라는 말로 표현한다. 이와 같이 BDP가 매우 큰 경우에는 윈도우도 그만큼 증가할 가능성이 높기 때문에, 현재 전송 중이고 아직 승인되지 않은 (outstanding) 패킷의 수 역시 큰 값을 가지게 된다. 이를 설명하기 위해서 다음의 그림 3에 한 예를 나타내었다.

그림 3에는 송신원과 수신원이 BDP가 16개 패킷 크기와 동일한 pipe로 연결된 상태에서 하나의 패킷 1이 손실된 경우 Fast Retransmit이 이루어지는 과정을 보여준다. 패킷 2~4에 의해서 패킷 1에 대한 세 개의 중복 승인을 전달하게 되고, 이를 수신한 송신원은 이를 Fast Retransmit에 의해서 재전송한다. 이 때, pipe 에는 여전히 전체 용량의 75%에 해당하는 12개의 패킷이 여전히 남아 있다. 즉, 남아있는 12개의 패킷들에 의해서 현재 pipe에 대한 정보를 충분히 얻을 수 있기 때문에 (다른 말로는 Clocked-State를 유지할 수 있기 때문에, 혹은 Probing이 가능하기 때문에), Fast Retransmit 후에 굳이 다시 Slow Start를 시작할 필요가 없다.

따라서 손실된 패킷을 Fast Retransmit에 의해 성공적으로 재전송되었다면 이 후의 전송은 ssthresh의 값으로 설정된 윈도우로 Congestion Avoidance 상태에서 계속된다. Fast Recovery 동안 송신원은 중복 승인 패킷

이 수신될 때 마다 윈도우의 크기를 1씩 증가시키고 아직 전송되지 않은 패킷이 윈도우에 포함되는 경우 전송된다. 이 패킷들에 의해서 pipe의 동기화 상태를 유지하는 것이 가능하다. 이 글의 나머지 부분에서는 손실된 패킷의 개수에 따른 TCP Tahoe, TCP Reno, TCP NewReno의 동작을 예를 통해서 설명한다.

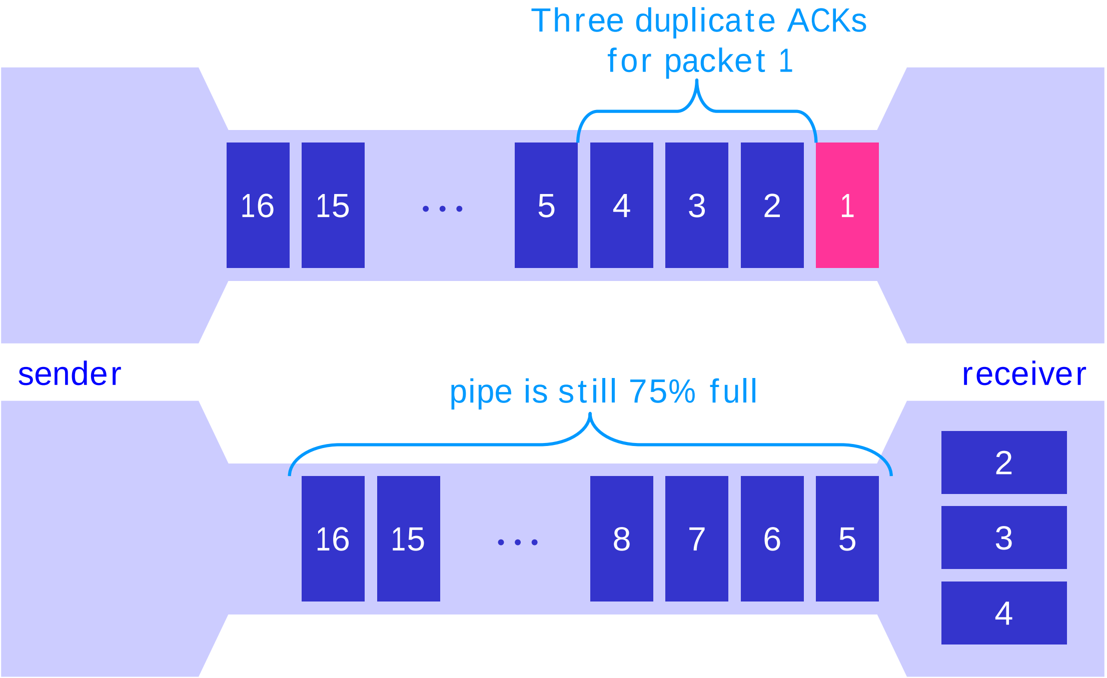
Three duplicate ACKs
for packet 1
16
15
...
5
4
3
2
1
sender
receiver
pipe is still 75% full
16
15
...
8
7
6
5
2
3
4

그림 3. Fast Recovery의 등장 배경

4. TCP Tahoe

다음 그림 4와 5에는 크기가 8인 윈도우에서 각각 1개, 2개의 패킷 손실이 발생했을 때 이들이 복구되는 과정 동안의 송신원의 윈도우의 변화를 보여준다.

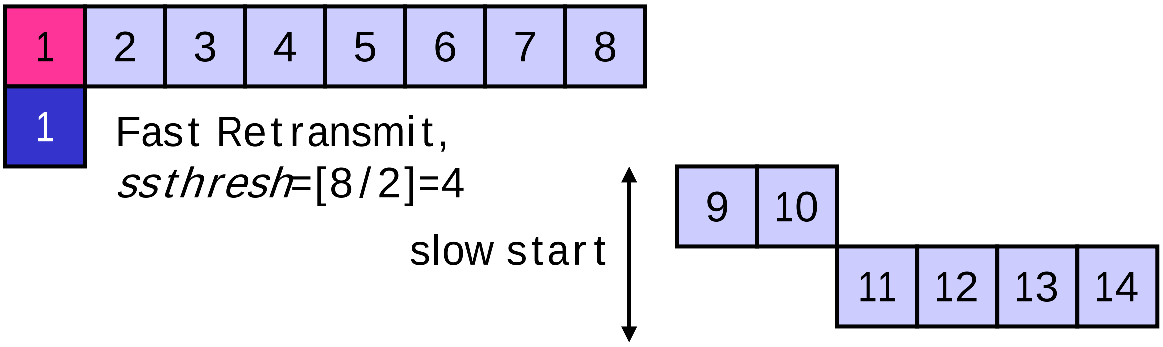
1
2
3
4
5
6
7
8
1
Fast Retransmit,
ssthresh=[8/2]=4
slow start
9
10
11
12
13
14

그림 4. 하나의 패킷 손실에 따른 TCP Tahoe의 동작

패킷 2~4에 의해서 손실된 패킷 1에 대한 세 개의 중복 승인을 수신할 때, 송신원은 Fast Retransmit에 의해 패킷 1을 재전송하고 ssthresh의 값을 현재 윈도우의 반인 4(=[8/2])로 설정한다. ([A]는 A의 정수 부분을 의미한다.) 이 후에 패킷 5~8에 의해서 네 개의 추가적인 중복 승인에 대해서 송신원은 아무런 동작을 하지 않는다. 패킷 1을 수신함으로써 수신원은 패킷 8까지의 전송이 정상적으로 이루어졌다는 것을 알리는 승인 패킷을 전달하는데, 이에 의해서 송신원의 윈도우의 크기는 Slow Start 상태에서 1 증가하여 패킷 9와 10이 전송된다. 패킷 9와 10에 의해서 두 개의 승인 패킷을 수신하면 윈도우의 크기는 4로 증가하는데, 이는 현재 설정된 ssthresh의 값과 동일하므로 패킷 15부터는 Congestion Avoidance 상태에서 전송된다.

그림 5에서와 같이 두 개의 패킷 손실이 발생한 경우에도 패킷 1을 Fast Retransmit에 의해서 재전송할 때까지 TCP Tahoe의 동작은 앞의 그림 4와 동일하다. 그러나 패킷 3이 손실되었기 때문에 수신원은 재전송한 패킷 1을 받았을 때 현재 패킷 2까지만 정상적으로 수신되었다는 것을 송신원에게 알려준다. 따라서, 윈도우는 패킷 3이 자신이 포함하는 첫 번째 패킷이 될 때까지 이동하고 (sliding), 이 때의 크기는 Slow Start 상태에서 1 증가하므로 패킷 3과 4를 전송한다. 여기서 중요한 점은 이 때의 송신원은 패킷 4가 손실되었는지 잘 전송되었는지에 대해서는 전혀 알 수 없다는 점이다. 그러므로, 이 예에서와 같이 패킷 4는 이미 정상적으로 전송되었음에도 불구하고 불필요하게 재전송된다. 패킷 3과 4를 수신한 수신원은 패킷 3에 대해서는 패킷 8까지 정상적으로 수신되었다는 승인 패킷을 전달하고, 패킷 4에 대해서는 동일한 내용의 중복 승인을 전달한다. 중복 승인은 윈도우의 크기를 증가시키지 않기 때문에, 패킷 3에 의한 정상적인 승인 패킷에 의해서만 윈도우의 크기가 1 증가하고 따라서 세 개의 패킷 9~11이 전송된다. 두 개 이상의 패킷 손실에 대해서도 TCP Tahoe의 동작은 동일하다.

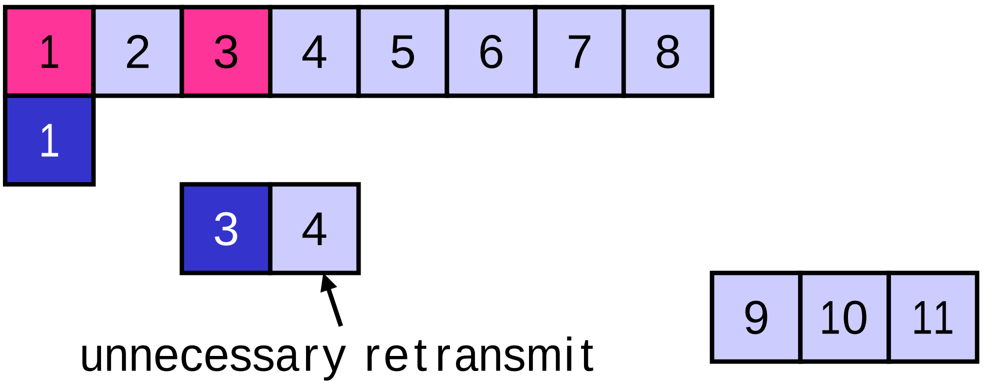

그림 5. 두 개의 패킷 손실에 따른 TCP Tahoe의 동작

5. TCP Reno

TCP Reno의 손실 복구 메커니즘에는 Fast Retransmit외에 앞에서 설명한 Fast Recovery 알고리듬이 포함되었다. 다음 그림 6은 하나의 패킷 손실에 따른 TCP Reno의 동작을 보여준다.

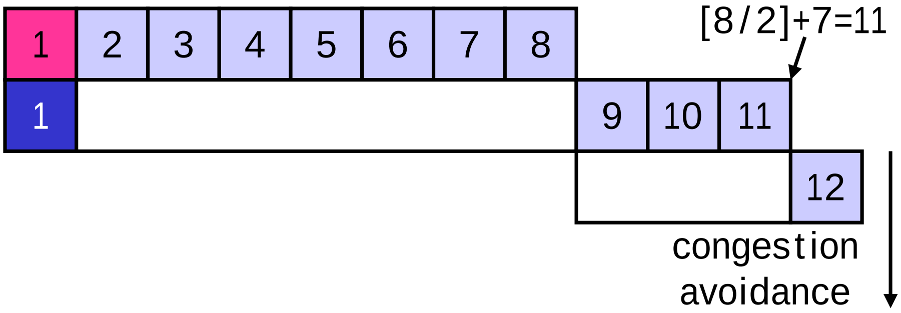

그림 6. 하나의 패킷 손실에 따른 TCP Reno의 동작

손실된 패킷 1이 Fast Retransmit에 의해서 재전송될 때까지의 동작은 TCP Tahoe와 동일하다. 이 후에 이어지는 Fast Recovery동안 송신원은 중복 승인 패킷이 수신될 때 마다 윈도우의 크기를 1씩 증가시킨다. (Fast Recovery동안 혼잡 윈도우 (cwnd)의 크기는 변화하지 않는다. 그렇기 때문에 Fast Recovery동안의 윈도우를 흔히 가용 윈도우 (usable window)라고 부르는데, 이 글에서는 그냥 윈도우라고 한다.) 이 예에서는 Fast Retransmit을 위해서 이미 수신한 세 개의 중복 승인을 포함해서 총 7개의 패킷 1에 대한 중복 승인 패킷들을 수신하게 되므로, 이들을 전부 수신했을 때의 윈도우의 크기는 11(=[8/2]+7)이다. 따라서 추가로 윈도우에 포함되는 세 개의 패킷들 9~11이 전송된다. 재전송한 패킷 1에 대해서 수신원은 패킷 8까지 정상적으로 수신했다는 승인을 전달하게 되는데, 이는 Fast Recovery를 종료하고 Congestion Avoidance가 시작되게 한다. 이 때 윈도우의 크기는 Fast Retransmit 당시 설정된 ssthresh의 값과 동일한 4이다. 이상 살펴본 바와 같이 TCP Reno는 손실된 패킷에 대한 Fast Retransmit이 성공적인 경우 다시 Slow Start를 시작할 필요가 없다는 장점이 있다.

그림 7에는 두 개의 패킷 손실에 대한 TCP Reno의 복구 과정을 나타내었다. 첫 번째 손실된 패킷 1이 Fast Retransmit에 의해 재전송되고 이 후의 Fast Recovery에서 추가적인 패킷들이 전송되는 과정은 앞에서 설명한 하나의 패킷 손실에 대한 경우와 동일하다. 단, 패킷 1에 대한 중복 승인 패킷의 수가 6개이기 때문에 Fast Recovery동안 윈도우는 최대 10까지 증가한다는 점에서 차이가 있다. 따라서 윈도우에 새롭게 포함된 두 개의 패킷 9~10이 전송된다. 재전송한 패킷 1에 의해서 전달되는 두 번째 손실된 패킷 3에 대한 승인에 의해 현재의 Fast Recovery는 종료되고 Congestion Avoidance가 시작된다. 이 때 윈도우의 크기는 첫 번째 Fast Retransmit 때 설정된 ssthresh의 값과 동일한데, 윈도우가 포함하는 네 개의 패킷은 이미 전송이 완료되었기 때문에 전송될 수 없다. 패킷 1의 재전송 이후에 전송된 두 개의 패킷에 의해서 패킷 2까지 정상적으로 수신되었다는 패킷 3에 대한 중복 승인이 발생한다. 그러나, 이 때 패킷 3을 Fast Retransmit에 의해서 재전송하기 위해서는 적어도 세 개의 중복 승인 패킷이 필요하다. 그러나 이 예에서는 두 개의 중복 승인 만이 수신되기 때문에 Fast Retransmit에 의한 재전송은 불가능하다. 결국, 두 번째 손실된 패킷 3은 타임 아웃이후 Slow Start상태에서 전송 된다. 패킷 3이 수신됨으로 인해 수신원은 패킷 10까지 정상적으로 수신되었다는 승인을 송신원에게 전달하고, 이는 윈도우의 크기를 1만큼 증가시키므로 두 개의 패킷 11과 12가 전송된다. RTO가 발생하면 ssthresh 값은 다시 반으로 감소하기 때문에 이 때의 ssthresh는 2로 설정되어 있고, 따라서 패

킷 13부터는 Congestion Avoidance상태에서 전송된다.

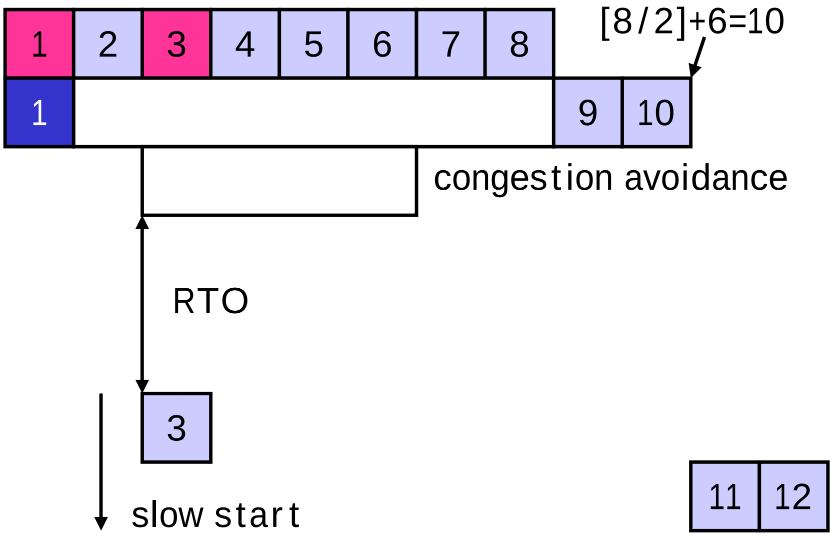

그림 7. 두 개의 패킷 손실에 따른 TCP Reno의 동작

앞의 예에서 살펴 본 바와 같이 하나의 윈도우 내에서 두 개 이상의 패킷이 동시에 손실되는 경우 TCP Reno는 이를 재전송에 의해서 복구하는 것이 불가능한 경우가 많다. 정확하게는 윈도우의 크기가 10보다 작을 때 두 개의 패킷 손실이 발생한 경우, 두 번째 패킷 손실은 항상 RTO 이 후에 전송되는 문제점이 있다. 이 문제점을 개선하기 위한 상당한 연구들이 진행되었고, 현재는 다음에 설명하게 될 TCP NewReno 혹은 Selective Acknowledgement (SACK) option에 의해서 거의 100% 해결하는 것이 가능하다.

6. TCP NewReno [3]

앞의 예를 통해서 여러 개의 패킷 손실이 발생했을 때 TCP Reno는 손실된 각각에 대해서 적어도 세 개 이상의 중복 승인을 수신해야 하는 점으로 인해서 이들의 재전송에 의한 복구 가능성이 낮다는 문제점에 대해서 알아보았다. TCP NewReno는 Fast Recovery의 동작을 약간 변경함으로써 이러한 문제를 해결할 수 있다. 이의 동작을 설명하기에 앞서, 부분 승인 (Partial Acknowledgement)에 대한 간단한 설명이 필요하다. 그림 7의 예에서 볼 수 있듯 Fast Retransmit에 의해서 재전송한 패킷 1에 의해서 수신되는 승인은 패킷 2까지만을 승인하는데 이는 이전에 전송한 모든 패킷, 즉 패킷 8까지를 승인하지 못하기 때문에 부분 승인이라고 한다. 송신원은 이미 자신이 패킷 8까지 전송을 완료했다는 것을 알고 있으므로 더 이상의 추가적인 패킷 손실이 없다면 재전송에 의해서 수신되는 승인 패킷은 적어도 패킷 8까지를 승인해야 한다. 그런데 손실된 패킷을 재전송했음에도 불구하고 패킷 8까지가 아닌 패킷 2까지만 정상적으로 전송되었다는 것은 또 다른 패킷 손실이 발생했다고 추정할 수 있는 근거가 된다. TCP NewReno는 이 사실을 기반으로 동작한다. 다음 그림 8에 두 개의 패킷 손실에 따른 TCP NewReno의 동작을 나타내었다. 하나의 패킷 손실에 따른 TCP NewReno의 동작은 TCP Reno와 동일하기 때문에 이에 대해서는 따로 설명하지 않는다.

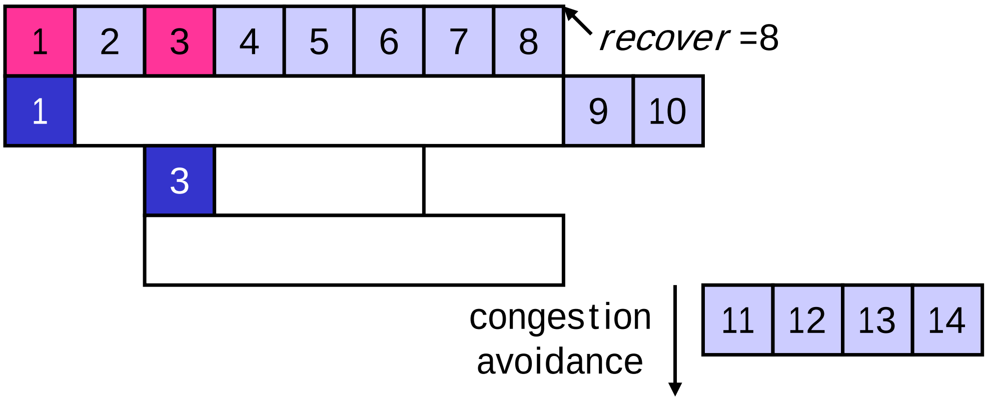

그림 8. 두 개의 패킷 손실에 따른 TCP NewReno의 동작

Fast Retansmit에 의해서 패킷 1을 재전송할 때, 송신원은 recover라는 변수에 가장 나중에 전송된 패킷의 순번 (Sequence Number)을 저장한다. 이 예에서는 패킷 8이 가장 나중에 전송되었으므로 8이 저장된다. 만약, 재전송한 패킷에 의해서 이 보다 낮은 패킷까지의 전송을 승인하는 승인 패킷이 도착한다면 이는 바로 부분 승인 패킷이다. 그림에서 볼 수 있듯, 패킷 3이 손실되었으므로 재전송한 패킷 1에 의해서 패킷 2까지만 정상적으로 전송되었다는 것을 알리는 부분 승인 패킷이 전달된다. 이 부분 승인 패킷을 수신한 송신원은 즉시 패킷 3을 재전송하고, 이 때 윈도우의 크기는 첫 번째 Fast Retransmit 때에 설정된 ssthresh의 값과 동일하게 설정한다. 패킷 9와 10에 의해서 발생하는 두 개의 중복 승인에 의해서 윈도우가 2만큼 증가하는 것은 TCP Reno의 Fast Recovery와 동일하다. 이 때, 윈도우에 포함되는 6개의 패킷 3~8들은 이미 전송이 완료되었으므로 더 이상의 패킷 전송은 없다. 재전송된 패킷 3에 의해서 송신원은 패킷 10까지 정상적으로 전송되었다는 승인을 수신하게 되는데 이 값은 recover에 저장된 값보다 크다. 따라서 송신원은 손실된 패킷들에 대한 복구가 끝났다는 것을 알게 되고, 현재의 Fast Recovery상태를 종료하고 Congestion Avoidance상태로 천이하게 된다. 이 때 윈도우의 크기는 ssthresh의 값과 동일한 4이다.

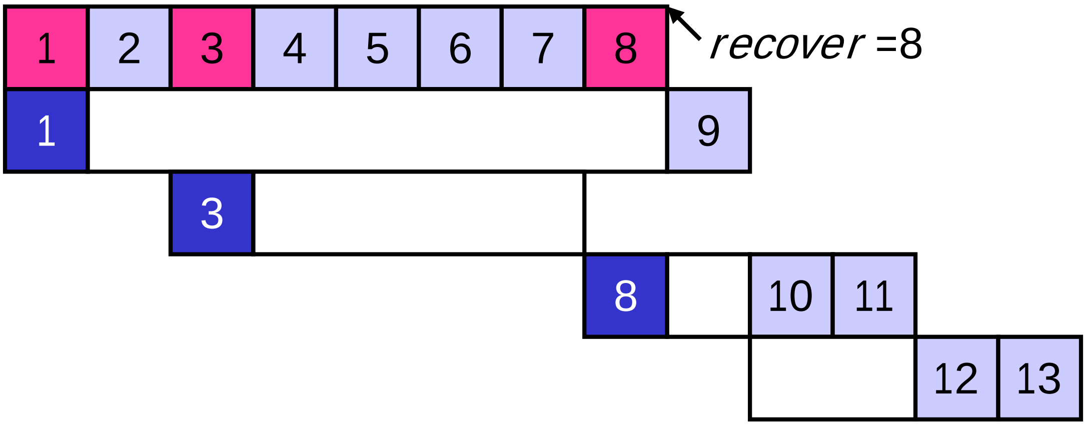

그림 9. 세 개의 패킷 손실에 따른 TCP NewReno의 동작

TCP NewReno의 TCP Reno와 가장 큰 차이점은 부분 승인을 수신했을 때 TCP NewReno는 Fast Recovery를 종료하지 않고 해당 손실된 패킷을 즉시 재전송한다는 점이다. 따라서, 추가적인 손실된 패킷에 대해서는 따로 세 개의 중복 승인을 받을 필요가 없어진다. 그리고, 패킷 손실의 개수에 관계없이 Fast Retransmit을 항상 한 번만 수행하기 때문에 손실된 패킷들의 복구과정 이후에는 패킷 손실이 발생하기 전 윈도우 크기의 반으로 Congestion Avoidance 상태에서 계속적인 전송이 가능하다. 그림 9는 세 개의 패킷 손실이 발생한 경우, TCP NewReno의 동작을 보여준다. 이의 동작은 앞에서 설명한 것과 동일하므로 생략하기로 한다.

7. Selective Acknowledgement (SACK) option [4]

앞서 말한 바와 같이 TCP의 승인 방식은 부정 승인 (Negative Acknowledgement)을 사용하지 않기 때문에 한 문제점이 있다. 이를 해결하기 위해서 제안된 것이 Selective Acknowledgement (SACK) option이다. SACK option을 사용하기 위해서는 송신원과 수신원이 모두 이를 지원해야 하고, SACK정보는 수신원이 전달하는 중복 승인 패킷의 헤더의 40 byte의 option 영역에 포함되어 송신원에게 전송된다. 다음 그림 10에 SACK option의 형식을 나타내었다.

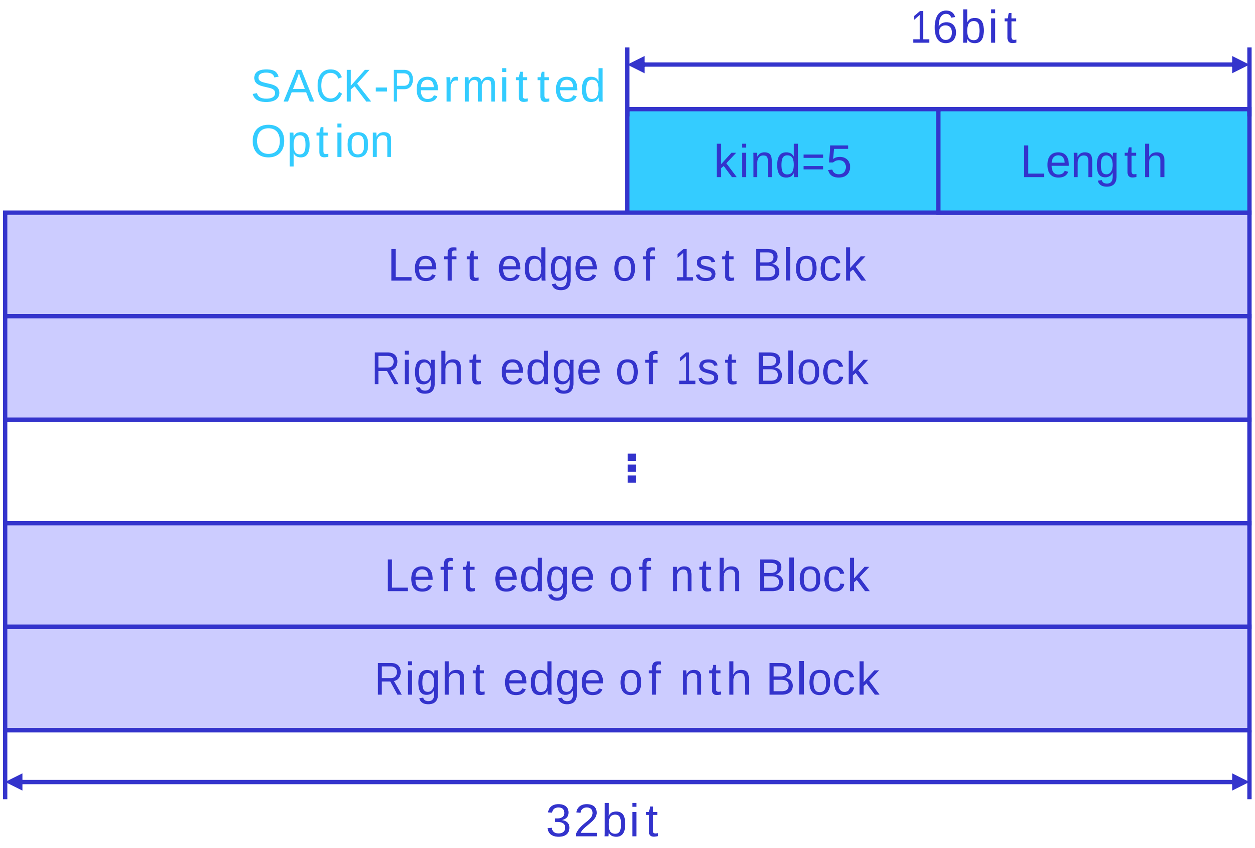
16bit
SACK-Permitted Option
kind=5
Length
Left edge of 1st Block
Right edge of 1st Block
⋮
Left edge of nth Block
Right edge of nth Block
32bit

그림 10. SACK option의 형식

SACK option은 현재 사용중인 option의 종류와 길이를 나타내기 위한 16bit의 SACK-Permitted option과 함께 사용된다. 나머지 영역에는 32bit의 블록의 왼쪽 경계 값과 오른 쪽 경계 값이 포함된다. 따라서, 40byte의 TCP option 영역에는 8*n+2<=40에 의해 최대 4개의 블록에 대한 정보가 포함될 수 있다. (만약, TCP Time-Stamp option이 사용되는 경우 3개의 블록으로 줄어든다.)

블록의 개념을 설명하기 위해서 그림 11에서와 같이 송신원이 윈도우의 크기가 8인 상태에서 1~8까지의 패킷을 전송하고 이들 중 패킷 1, 패킷 3, 패킷 7이 손실된 경우를 생각해보자. 손실되지 않은 5개의 패킷들에 의해서는 모두 패킷 1에 대한 중복 승인 즉, 패킷 1을 기다리고 있다는 중복 승인들이 발생한다. 패킷 2가 수신되었을 때, 수신원은 패킷 1에 대한 중복 승인과 함께 첫 번째 블록인 자신의 왼쪽 경계 (Left Edge; LE)와 오른쪽 경계 (Right Edge; RE)의 byte를 전달한다. 패킷 3이 수신되었을 때 전달되는 중복 승인의 SACK 정보에는 두 개의 블록에 대한 정보가 포함되는데, 이 때 항상 새로 형성된 블록이 첫 번째 블록이 된다. 따라서 첫 번째 블록은 현재 패킷 4 자신이 되므로 이의 왼쪽 경계와 오른쪽 경계의 byte를 전달한다. 이 후 패킷 5와 6은 블록의 수를 증가시키지 않고 첫 번째 블록의 오른쪽 경계만을 계속 증가시킨다. 마지막으로 패킷 8이 수신되면 패킷 7이 손실되었기 때문에 새로운 첫 번째 블록이 형성되고, 이에 의한 마지막 중복 승인에는 모두 세 개의 블록에 대한 정보가 포함된다. 따라서 패킷 8까지 모두 수신했을 때의 수신원 내의 큐는 그림의 오른쪽에 나타난 것과 같다. 세 개의 블록이 세 개의 패킷 손실로 (이를 일반적으로는 hole이라고 한다.) 구분되어 있는 것을 볼 수 있다.

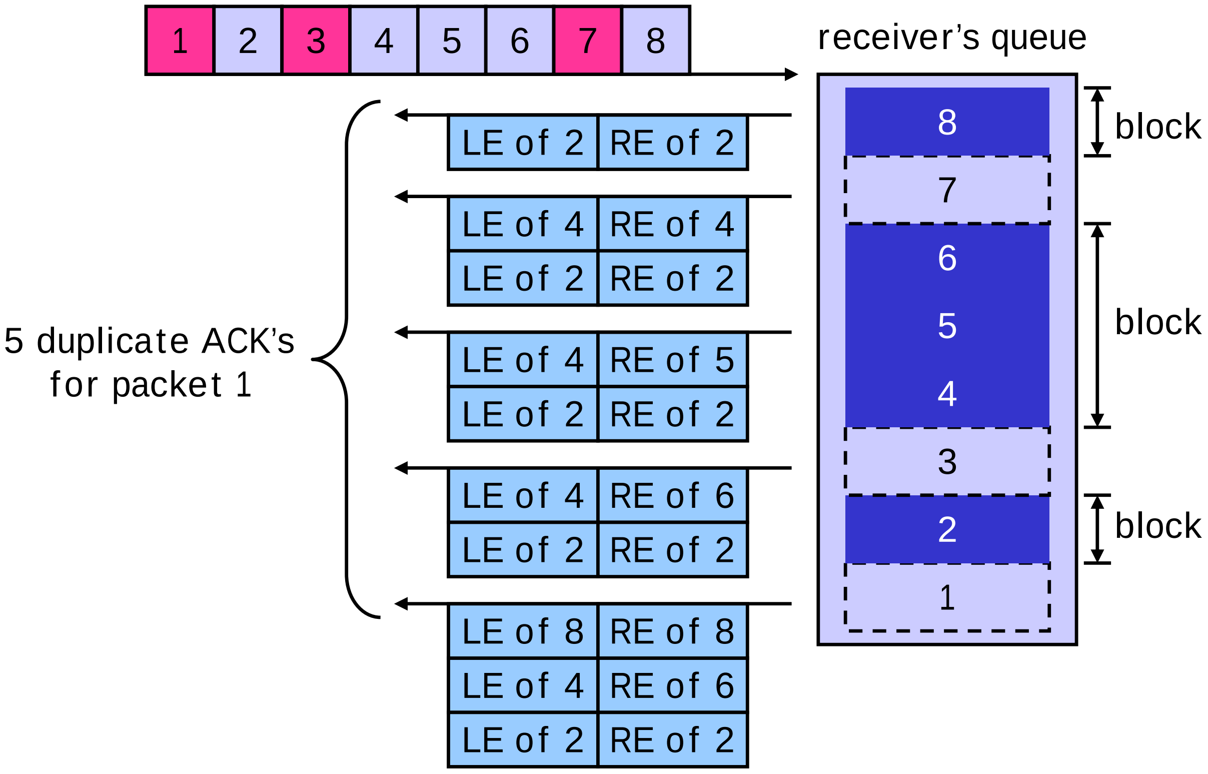
1
2
3
4
5
6
7
8
receiver's queue
LE of 2
RE of 2
LE of 4
RE of 4
LE of 2
RE of 2
LE of 4
RE of 5
LE of 2
RE of 2
LE of 4
RE of 6
LE of 2
RE of 2
LE of 8
RE of 8
LE of 4
RE of 6
LE of 2
RE of 2
5 duplicate ACK's
for packet 1
8
7
6
5
4
3
2
1
block
block
block

그림 11. SACK option의 동작 예

앞에서 살펴본 SACK option은 말 그대로 하나의 option일 뿐 이를 혼잡 제어에 적용하는 방법은 다양할 수 있다. 이 글에서는 가장 널리 알려진 Sack1 (이는 ns에 구현되어 있다.)을 기준의 SACK option을 사용하는 TCP의 손실 복구 과정에 대해서 설명하도록 한다.

Sack1은 현재 전송이 진행중인 패킷의 대략적인 개수를 세기 위해서 pipe라는 변수를 사용한다. Fast Recovery동안 중복 승인이 도착할 때마다 pipe 값은 1씩 감소되는데, 이는 중복 승인이 도착했다는 것은 전송 중이던 패킷들 중 하나가 전송이 완료되었다는 것을 의미하기 때문이다. 부분 승인이 도착할 때는 pipe 값을 2씩 감소시킨다. 이는 전송 중이던 패킷들 가운데 하나는 전송이 완료되었고 하나는 손실되었다는 것을 의미하기 때문이다. 줄어든 pipe 값이 cwnd보다 작아지면 추가적인 패킷을 전송하는 것이 가능하고, 이때 pipe 값은 1씩 증가한다.

하나의 패킷 손실에 대한 Sack1의 동작은 TCP Reno와 동일하기 때문에, 다음 그림 12에는 두 개의 패킷 손실에 따른 Sack1의 동작을 나타내었다. 첫 번째 손실된 패킷이 Fast Retransmit에 의해 재전송되기까지의 동작은 동일하다. 세 개의 중복 승인 패킷을 수신했을 때 송신원은 패킷 1을 재전송하면서 cwnd=4, pipe는 현재 윈도우의 크기에서 세 개의 중복 승인을 받았으므로 5로 설정한다. 이 후 패킷 6, 7, 8에 의해서 세 개의 중복 승인을 추가로 수신하게 되는데 이 때 pipe는 1씩 감소한다. 5번째 중복 승인을 받았을 때 pipe는 3으로 감소하므로 cwnd보다 작은 값을 가지게 되어 하나의 패킷을 전송하는 것이 가능해진다. 그런데, 이 때 이미 송신원은 패킷 3 이 후에 전송된 패킷들에 의해서 발생하는 중복 승인들의 SACK정보를 통해서 패킷 3도 손실되었다는 것을 알 수 있다. 따라서 이 때 손실된 패킷 3을 우선해서 보내게 된다. 6번째 중복 승인을 받았을 때 다시 pipe는 3으로 감소하는데 더 이상 손실된 패킷은 없기 때문에 패킷 9를 전송한다. 재전송한 패킷 1에 의해서 송신원은 패킷 3에 대한 부분 승인을 받게 되는데 이는 pipe를 2만큼 감소시킨다. 더 이상의 패킷 손실은 없기 때문에 두 개의 패킷 10~11이 전송된다. 바로 뒤에 수신되는 재전송한 패킷 3에 의한 승인은 패킷 8까지 정상적으로 전송되었다는 것을 알려주기 때문에 이 시점에서 Fast Recovery는 종료되고 다시 Congestion Avoidance 상태에서 계속적인 전송이 진행된다. 이 때 윈도우의 크기는 이전에 설명했던 TCP들과 마찬가지로 4이다.

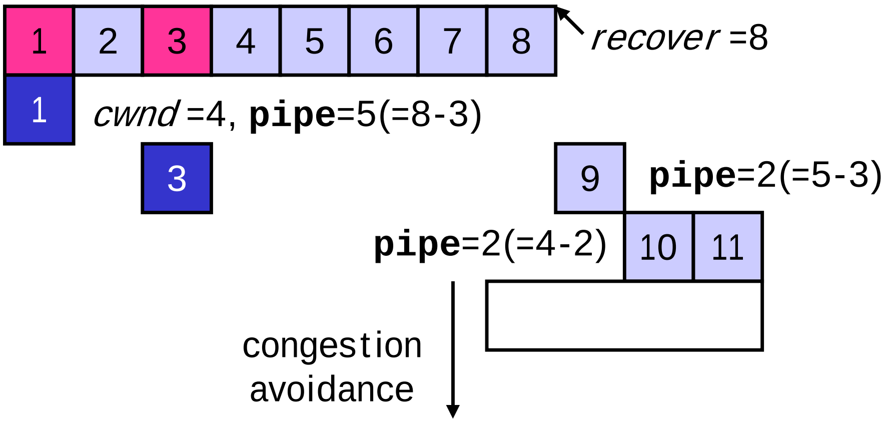

그림 12. 두 개의 패킷 손실에 따른 TCP SACK (Sack1)의 동작

다음 그림 13은 세 개의 패킷 손실에 따른 Sack1의 동작을 보여준다. 앞의 설명을 토대로 이의 동작을 쉽게 이해할 수 있다. 패킷 1을 Fast Retransmit에 의해 재전송한 이 후, 패킷 1에 대한 모든 중복 승인을 받았을 때 pipe는 3이므로 하나의 추가적인 패킷을 전송할 수 있는데, 이미 패킷 3이 손실되었다는 것을 알고 있기 때문에 이를 먼저 전송한다. 한 RTT 후에는 재전송한 패킷 1과 패킷 3에 의해서 두 개의 중복 승인이 수신되므로 pipe는 0(4-4)으로 감소하여 네 개의 패킷을 전송할 수 있게 된다. 이미 이전의 패킷 8에 의한 중복 승인의 SACK 정보를 통해서 패킷 7이 손실되었다는 것을 알고 있으므로 이의 재전송과 세 개의 새로운 패킷들 9~11이 전송된다. 이 후 재전송한 패킷 7에 의해서 패킷 8까지 정상적으로 전송되었다는 승인을 수신하면 Congestion Avoidance가 시작되고 이 때의 윈도우의 크기는 4이다.

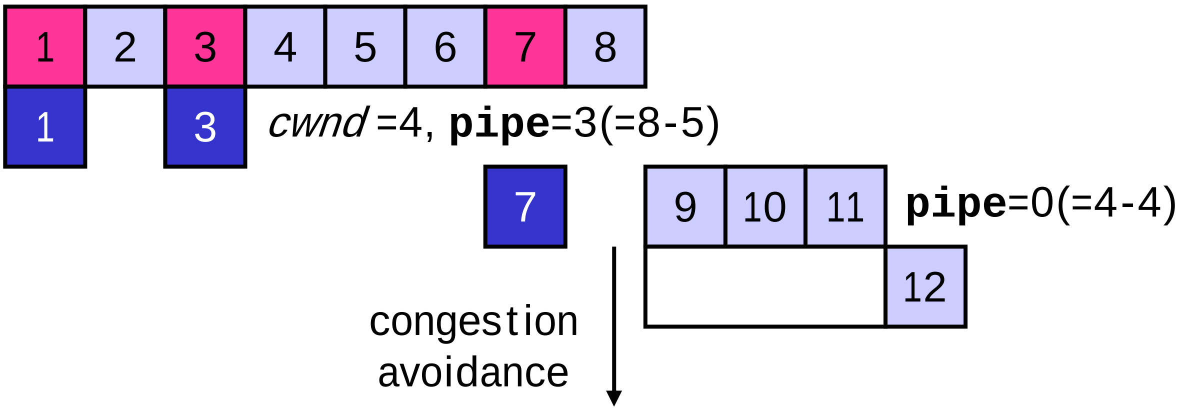

그림 13. 세 개의 패킷 손실에 따른 TCP SACK (Sack1)의 동작

8. 요약

본 글에서는 TCP 혼잡 제어에 구현되어 있는 패킷 손실 복구 메커니즘에 대해서 알아보았다. 손실 복구의 성능은 TCP의 성능에 전반적인 영향을 미치기 때문에 이 기능은 거듭 수정되어 왔고, 흔히 TCP Reno, TCP NewReno, TCP SACK등으로 불리고 있다. 본 글에서는 패킷 개수에 따른 각 TCP의 동작을 예를 통해서 살펴봄으로써 좀 더 쉽게 이해할 수 있다. 이에 대한 자세한 내용은 다음의 참고 문헌을 주로 참조하였다.

[참고문헌]

[1] V. Jacobson, Congestion Avoidance and Control, ACM SIGCOMM 88, pp. 314-329, 1988.
[2] V. Jacobson, Modified TCP Congestion Avoidance Algorithm, 1990,
[Online] ftp://ftp.ee.lbl.gov/vanj.90apr30.txt.
[3] Janey C. Hoe, Improving the Start-Up Behavior of a Congestion Control Scheme for TCP,
ACM SIGCOMM 96, 1996.
[4] K. Fall and S. Floyd, Simulation-based Comparisons of Tahoe, Reno, and SACK TCP,
ACM SIGCOMM Computer Communication Review (CCR), vol. 26, no. 3, pp. 5-21, 1996.