Reno_custom.c 코드는 tcp 통신의 congestion을 관리하는 프로그램입니다.

이 코드를 바꾸기 앞서 먼저 내용을 분석하도록 하겠습니다.

tcp_reno_init(struct sock *sk):

TCP Reno의 변수들을 초기화합니다. snd_cwnd를 1로 설정하고, snd_ssthresh TCP INFINITE SSTHRESH으로 설정됩니다. 이 변수는 이후 slow start를 멈추는 기준값이 됩니다.

tcp_reno_ssthresh(struct sock *sk):

congestion이 발생할 때, 윈도우의 크기를 절반으로 감소시킵니다.

tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked):

congestion avoid 단계를 관리합니다. 윈도우 크기가 slow start 기준값 아래인지 여부에 따라 혼잡 window 크기를 다르게 증가시킵니다. 기준값 이하인 경우, tcp_slow_start 함수를 통해 window값을 변화시키고 이상이면 cong_avoid_ai 함수로 window값을 변화합니다.

tcp_reno_undo_cwnd(struct sock *sk):

필요한 경우 window의 변경 사항을 취소하여 이전 상태로 되돌립니다..

tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked) 함수 내부에 있는 **tcp_cong_avoid_ai**와 **tcp_slow_start** 함수를 분석하겠습니다. 둘 다 reno custom.c에 존재하지 않고 net/ipv4/tcp_cong.c 파일에 존재하는 함수입니다. 하지만 위 알고리즘의 실행에 가장 중요한 역할을 합니다.

```
__bpf_kfunc u32 tcp_slow_start(struct tcp_sock *tp, u32 acked)
{
    u32 cwnd = min(tcp_snd_cwnd(tp) + acked, tp->snd_ssthresh);

    acked -= cwnd - tcp_snd_cwnd(tp);
    tcp_snd_cwnd_set(tp, min(cwnd, tp->snd_cwnd_clamp));

    return acked;
}
```

Tcp slow start 함수입니다. 초반에 급격한 성장을 위한 코드로 ack의 개수만큼 window의 크기를 증가시킵니다. 마지막에 window가 한계치를 넘어가지 않도록 제한을 두는 함수를 가지고 있습니다.

```
__bpf_kfunc void tcp_cong_avoid_ai(struct tcp_sock *tp, u32 w, u32 acked)
{
    /* If credits accumulated at a higher w, apply them gently now. */
    if (tp->snd_cwnd_cnt >= w) {
        tp->snd_cwnd_cnt = 0;
        tcp_snd_cwnd_set(tp, tcp_snd_cwnd(tp) + 1);
    }

    tp->snd_cwnd_cnt += acked;
    if (tp->snd_cwnd_cnt >= w) {
        u32 delta = tp->snd_cwnd_cnt / w;

        tp->snd_cwnd_cnt -= delta * w;
        tcp_snd_cwnd_set(tp, tcp_snd_cwnd(tp) + delta);
    }
    tcp_snd_cwnd_set(tp, min(tcp_snd_cwnd(tp), tp->snd_cwnd_clamp));
}
```

Tcp_cong_avoid_ai 함수입니다. Window 크기가 특정 값을 넘어가는 순간, window 크기를 1씩 증가시킵니다.

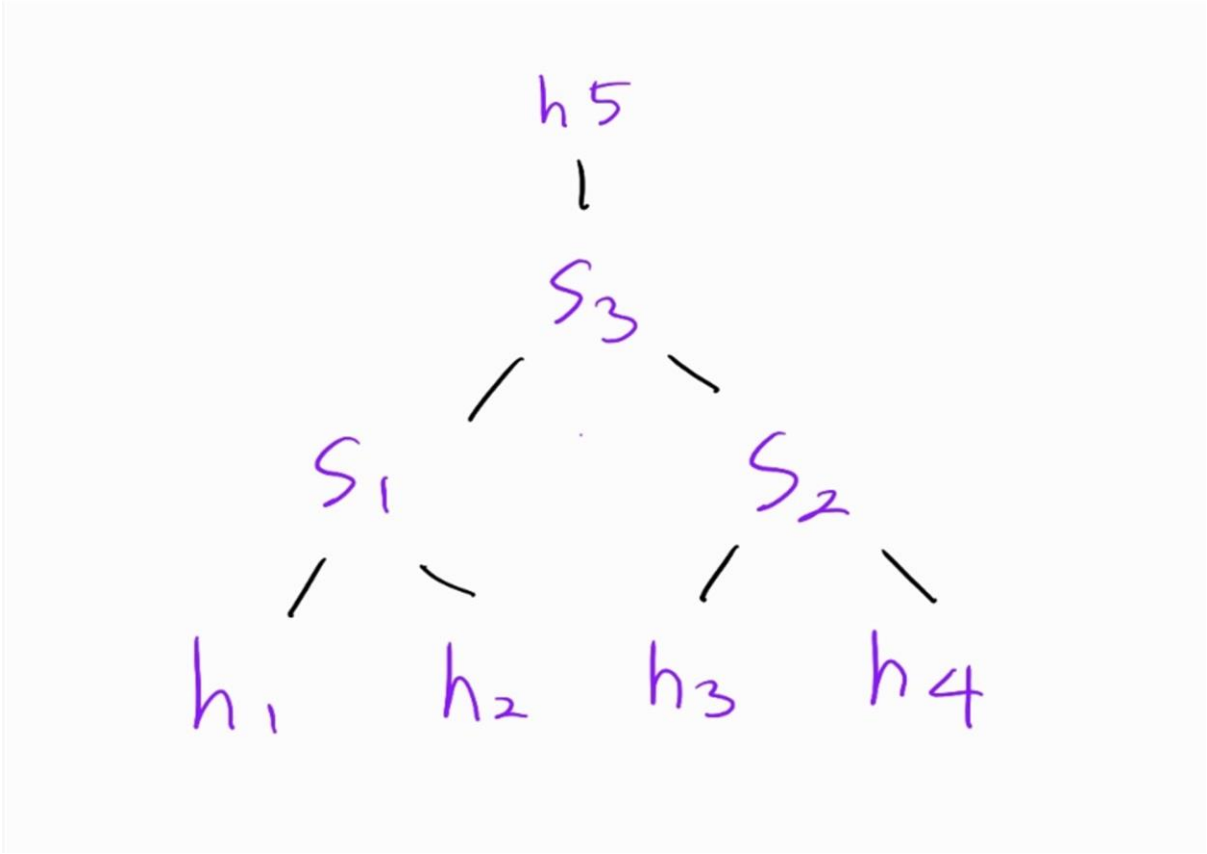
Network topology 로는 아래와 같은 형태의 네트워크를 설정했습니다.

다수의 호스트가 하나의 호스트로 모이는 형태로 제작했습니다. 이렇게 하나의 link 에 데이터가 집중되는 상황이 congestion을 재현하기 좋다고 생각했습니다.

실험의 특정은 h1, h2, h3, h4와 h5 사이의 데이터를 기록했습니다. 각 연결의 최대 값은 100Gbit이며 iperf함수로 bandwidth(throughput)를 측정했습니다. Ping 또한 내장된 함수로 5회씩 측정했습니다.

Reno_custom 테스트는 5회씩 실행했으며 값의 편차가 가장 큰 경우와 작은 경우는 제외했습니다. 실행 상황에 따른 데이터의 편차가 무시 못할 정도로 크게 측정이 되어서 이런 선택을 했습니다. 즉 알고리즘 당 3회의 값을 기록했습니다.

알고리즘의 지표는 bw의 평균, rtt의 평균, bw의 표준편차의 평균, rtt의 표준편차의 평균입니다. (3회의 실험 각각 표준편차를 측정한 후, 그것의 평균을 구했습니다.). Bw로 link usage, rtt로 latency, 표준편차로 fairness를 표현하고자 했습니다.



우선 비교의 기준이 될 알고리즘부터 측정했습니다. 기본적으로 주어진 reno_custom.c 그대로 성능을 테스트했습니다. 결과는 아래와 같습니다.

reno custom default			
bw	rtt		
23	4.25	bw	23.06667
24.8	4.76	rtt	4.271667
23.6	4.25	bw-sd	1.378842
24.9	3.48	rtt-sd	0.472033
23.1	4.19	bw-sd	0.928709
24.2	4.66		1.687947
20.2	4.38		1.519868
22.4	5.05		
		rtt-sd	0.52792
23.9	3.55		0.37372
23.7	3.79		0.514458
20.6	4.73		
22.4	4.17		

reno default			
bw	rtt		
26.7	3.09	bw	26.69167
26.6	3.41	rtt	3.3175
26.4	3.97	bw-sd	0.811201
25.7	2.74	rtt-sd	0.442087
25	2.75	bw-sd	0.450925
27.4	2.7		1.297112
26.7	2.56		0.685565
28	3.31		
		rtt-sd	0.52239
27	3.47		0.329949
27.9	3.65		0.47392
26.5	4.52		
26.4	3.64		

왼쪽이 reno_custom.c의 성능, 오른쪽이 reno 알고리즘의 성능입니다.

재미있는 점은 두 알고리즘이 동일한데도, reno의 성능이 훨씬 좋다는 것입니다. Bw 은 10%, rtt 는 25%, bw fairness는 40%정도 성능이 좋습니다. 단순히 함수를 그대로 가져왔음에도 성능이 차이가 납니다.

제 추측으로는 tcp 통신의 특성상, window 크기를 조절하기 위해 짧은 시간에 다수의 함수를 호출하는데, reno_custom을 추가적으로 호출하면서 생기는 overhead에 의해 발생된 것으로 생각했습니다.

다음으로 reno_custom.c 내부의 알고리즘을 극단적으로 변화시키면서 실험을 했습니다. 알고리즘이 성능에 얼마나 큰 영향을 줄 수 있나 보기위해 이런 시도를 했습니다.

reno1	무제한 증가		
bw	rtt		
18.6	6.11	bw	18.84167
20.3	3.219	rtt	3.347083
19.1	2.301	bw-sd	0.609638
19.3	2.421	rtt-sd	1.032591
18.6	2.861	bw-sd	0.713559
18.5	2.711		0.408248
17.8	4.327		0.707107
18.7	4.052		
		rtt-sd	1.778787
18.1	3.363		0.82034
19.7	2.749		0.498645
18.4	3.553		
19	2.498		

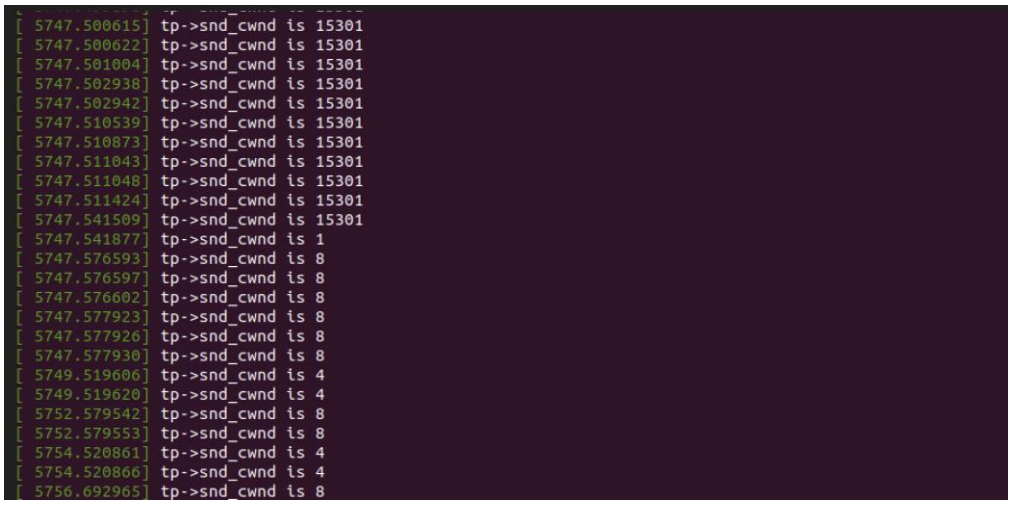
reno2	항상 cwnd =2		
bw	rtt		
19.1	2.848	bw	22.025
21.1	2.845	rtt	3.07225
21.8	2.948	bw-sd	0.812144
23.2	3.037	rtt-sd	0.34141
22.5	3.449	bw-sd	1.706849
22.4	2.402		0.25
22.7	3.572		0.479583
22.1	3.382		
		rtt-sd	0.091799
21.7	2.976		0.538611
22.8	3.015		0.39382
22.3	2.736		
22.6	3.657		

왼쪽은 window의 크기를 멈추지 않고 무제한적으로 증가시킬 경우의 결과이고 오른쪽은 window의 크기가 항상 2인 경우입니다. 둘 다 성능이 나빠지는 것을 볼 수 있었습니다. Window의 크기가 항상 2인 경우는 bw의 성능은 나빠졌지만 rtt가 감소한 것을 보아 lantency 측면에서는 꽤 성능이 증가했습니다. Rtt의 표준편차도 감소한 것을 보아, window의 크기를 감소시키면서 link usage 대신 fairness도 좋아진 것을 볼 수 있었습니다.

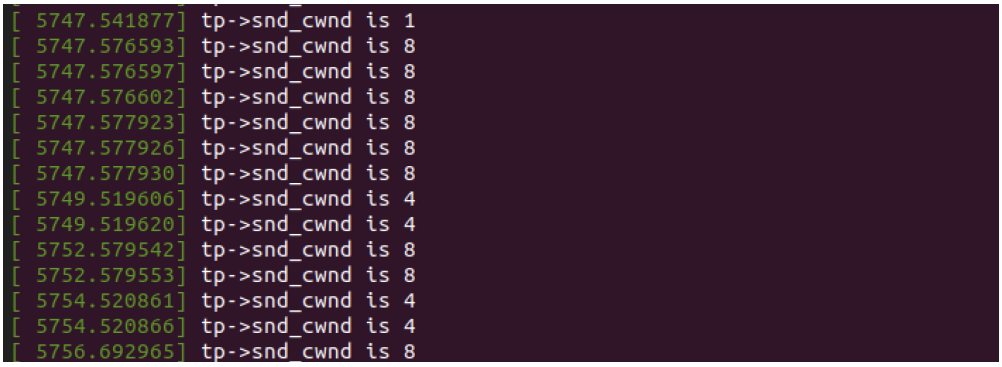
정말 극단적인 알고리즘을 줘도 의외로 성능은 기대한 만큼 악화되지 않았습니다.

다음으로는 dmesg을 이용한 window의 사이즈를 분석해봤습니다.

아래는 reno_custom.c의 window 크기를 출력한 결과의 일부입니다.



실행을 했을 때, 다음과 같이 초반부터 10000이 넘는 경우가 대다수였습니다. 초반에 통신 과정에서 급격하게 window의 사이즈가 증가해서 발생하는 일로 생각했습니다. 그 뜻은 초반부터 window의 사이즈를 크게 적용하면 더 성능이 좋아지지 않을까 생각을 했습니다.



또 다른 특징으로는 10000이 넘는 window의 크기가 유지되다가 급격하게 window의 크기가 감소하고 다시 증가하지 않는다는 것이었습니다. 아마도 통신간의 congestion이 정점에 도달해서 계속 window 사이즈를 1씩 증가시키다 절반씩 감소되는 단계로 보입니다.

이러한 현상을 바탕으로 초반에 급격한 증가와 후반의 급격한 감소를 더 효율적으로 고치면 reno_custom의 성능을 개선시킬 수 있다고 추측했습니다.

위의 가설을 바탕으로 실험을 해보았습니다. 초기 window 를 5000, 10000 으로 설정하고 성능을 비교해보았습니다.

reno3_1	초기 cwnd 5000		
bw	rtt		
23.4	3.42	bw	23.025
22.6	2.81	rtt	2.931667
23.5	3.04	bw-sd	0.958954
20.7	3	rtt-sd	0.265808
23.3	3.07	bw-sd	1.297433
23	3.02		1.357694
24.6	2.67		0.221736
21.3	2.82		
		rtt-sd	0.255522
23.6	2.79		0.184842
23.4	3.35		0.35706
23.7	2.6		
23.2	2.59		

reno3_2	초기 cwnd 10000		
bw	rtt		
23.3	2.6	bw	23.34167
25.1	3.28	rtt	3.306667
24.4	3.08	bw-sd	1.476871
23.7	2.35	rtt-sd	0.940787
21.8	2.56	bw-sd	0.7932
22.5	3.12		1.57586
20.9	3.29		2.061553
24.6	2.11		
		rtt-sd	0.427502
20.5	2.45		0.539321
23.9	3.11		1.855539
24.1	5.32		
25.3	6.41		

결과는 초기 window 의 값이 10000 인 경우가 성능이 더 좋았습니다. 초반에 window 의 크기가 10000 이상이 나온 것을 더 잘 반영해서 그런 것 같습니다. Fairness 측면에서는 예상대로 5000 인 경우가 더 좋았지만 10000 의 경우도 여전히 초기의 reno_custom 보다는 좋은 경향을 보여서 bw 측면에서 성능이 좋은 10000 이 더 좋은 알고리즘이라 판단했습니다.

다음으로는 내부 변수들을 조금씩 변화시키면서 성능을 비교했습니다. 급격한 감소를 비교하기 위한 실험을 시행했습니다.

reno3_2	초기 cwnd 10000			reno3	초기 cwnd 10000 감소 1/4		
bw	rtt			bw	rtt		
23.3	2.6	bw	23.34167	24.3	2.705	bw	23.425
25.1	3.28	rtt	3.306667	21.8	2.934	rtt	3.194917
24.4	3.08	bw-sd	1.476871	24.9	3.09	bw-sd	1.809895
23.7	2.35	rtt-sd	0.940787	24.1	2.41	rtt-sd	1.284198
21.8	2.56	bw-sd	0.7932	24.3	2.64	bw-sd	1.359841
22.5	3.12		1.57586	23.7	2.32		1.209339
20.9	3.29		2.061553	22.5	2.44		2.860507
24.6	2.11			21.6	3.39		
		rtt-sd	0.427502			rtt-sd	0.295664
20.5	2.45		0.539321	19.3	8.7		0.480165
23.9	3.11		1.855539	24.4	2.95		3.076766
24.1	5.32			25.8	2.37		
25.3	6.41			24.4	2.39		

오른쪽의 실험은 초기의 cwnd 설정값을 10000 으로 설정하는 대신 감소시킬 때, 1/2 이 아닌 1/4 씩 감소시키는 알고리즘을 적용했습니다.

실험 결과 약간의 bw 상승, 10%대의 rtt성능 상승이 있었지만 bw의 표준편차가 50%, rtt의 표준편차는 300% 가량 악화되는 것을 볼 수 있었습니다. 초반에 window의 값을 높게 잡은 대신 급격하게 감소시키는 방법은 예상대로 link utilization, lantency를 위해 fairness를 크게 희생하는 모습을 보였습니다.

Fairness 측면에서 무시 못할 성능하락을 보였지만 3번째 실험에서 하나의 예외적인 값이 전체 평균에 영향을 미치는 것으로 보입니다. 그 값을 예외라 판단할 경우 성능이 향상되었다고 보아도 될 것 같습니다.

다음 실험은 slow start phase 에서 더 공격적으로 증가하는 방법을 실험했습니다.

reno custom default			
bw	rtt		
23	4.25	bw	23.06667
24.8	4.76	rtt	4.271667
23.6	4.25	bw-sd	1.378842
24.9	3.48	rtt-sd	0.472033
23.1	4.19	bw-sd	0.928709
24.2	4.66		1.687947
20.2	4.38		1.519868
22.4	5.05		
		rtt-sd	0.52792
23.9	3.55		0.37372
23.7	3.79		0.514458
20.6	4.73		
22.4	4.17		

reno4	thresh 보다 작을 때 3/2 증가		
bw	rtt		
25	2.65	bw	23.03333
24.1	2.9	rtt	2.8075
22.9	3.21	bw-sd	0.651581
23.7	2.33	rtt-sd	0.329885
24.4	2.99	bw-sd	0.873212
22.9	2.57		0.818535
24.1	3		0.262996
22.8	2.97		
		rtt-sd	0.373486
21.6	2.43		0.208706
21.4	2.53		0.407462
22	2.77		
21.5	3.34		

오른쪽의 실험은 slow start 부분, 즉 thresh 의 값보다 window 가 작을 때 window 를 ack * 3/2 만큼 증가시키는 실험을 했습니다.

오른쪽의 경우 bw에는 큰 변화가 없었고 latency와 fairness 지표에서 성능이 향상되었습니다. 특히 bw-fairness가 크게 증가했습니다. 제 생각으로는 threshold 이하의 구간에서 단기간에 증가시키면서 avoid congestion 단계를 일찍 들어가는게, 다른 통신에게 자유롭게 window를 증가시킬 여유를 주어서 fairness에 긍정적인 영향을 준 것으로 생각합니다.

다음 실험은 reno 와 reno_custom 의 차이를 줄이기 위해 실행한 실험입니다.

Reno_custom 에 없는 함수들을 파일 내로 옮기고 실험을 했습니다. 이를 new_reno.c, n_reno 라고 부르겠습니다.

reno custom default				new_reno_default			
bw	rtt			bw	rtt		
23	4.25	bw	23.06667	24.7	5.54	bw	23.625
24.8	4.76	rtt	4.271667	21.9	3.2	rtt	3.943333
23.6	4.25	bw-sd	1.378842	23.4	5.22	bw-sd	0.899456
24.9	3.48	rtt-sd	0.472033	23.5	2.84	rtt-sd	0.923491
23.1	4.19	bw-sd	0.928709	23.3	3.61	bw-sd	1.147098
24.2	4.66		1.687947	24.3	3.33		1.252664
20.2	4.38		1.519868	23.2	4.64		0.298608
22.4	5.05			21.3	3.34		
		rtt-sd	0.52792			rtt-sd	1.376663
23.9	3.55		0.37372	24.6	2.99		0.620376
23.7	3.79		0.514458	24.4	4.14		0.773434
20.6	4.73			24.8	4.82		
22.4	4.17			24.1	3.65		

두 실험의 차이점은 reno_custom.c 내부에 tcp_slow_start() 와 tcp_cong_avoid_ai 함수가 존재하냐의 차이입니다. Net/ipv4/tcp_cong.c 내부에 있던 함수를 reno_custom 에 넣어서 따로 호출할 필요가 없도록 제작했습니다. 함수는 전혀 바꾸지 않아도 거의 모든 면에서 성능이 증가했습니다.

다음 실험은 new_reno.c 와 이전의 방식을 동시에 적용한 결과를 측정했습니다.

성능이 좋았던 reno3 과 reno4 를 적용했습니다.

n_reno3	cwnd 10000 감소폭 1/4		
bw	rtt		
24.3	3.52	bw	23.79167
24.1	3.3	rtt	3.599167
22.7	3.18	bw-sd	0.674366
23.2	4.94	rtt-sd	0.569459
24.3	3.11	bw-sd	0.754431
24	3.42		0.826136
22.5	3.93		0.442531
24.1	3.67		
		rtt-sd	0.815578
23.6	2.74		0.350274
24.5	3.75		0.542525
23.8	3.97		
24.4	3.66		

n_reno4	thresh 보다 작을 때 3/2 증가		
bw	rtt		
25.3	2.99	bw	24.21667
24.4	4.56	rtt	3.563333
23.3	4.21	bw-sd	1.207235
22.7	3.33	rtt-sd	0.670424
24.6	3.22	bw-sd	1.155783
25.4	3.42		1.885692
21.8	3.9		0.58023
26.1	2.64		
		rtt-sd	0.734773
24.5	2.69		0.521632
24.7	4.28		0.754868
23.4	4.19		
24.4	3.33		

예상대로 두 실험 모두 이전의 reno3 reno4 보다는 더 좋은 성능을 보였습니다.

특이한 점은 이전 실험결과에서는 reno3 이 더 좋은 bw, 낮은 rtt, fairness 를 보였다면 이번 실험은 반대로 결과가 나왔습니다. 그리고 성능의 차이가 더 벌어진 것으로 보입니다. 결론적으로 지금까지 n_reno4 가 가장 좋은 성능이 좋습니다.

n_reno5		3, 4 합치기	
bw	rtt		
22.3	3.66	bw	24.16667
23.5	3.55	rtt	3.226667
25.4	3.46	bw-sd	2.044941
25	3.03	rtt-sd	0.802686
21.9	3.93	bw-sd	1.424781
25.4	4.21		1.931105
24.4	0.59		2.778939
26.4	3.45		
		rtt-sd	0.275741
20.1	2.57		1.666483
26.6	3.68		0.465833
23.9	3.35		
25.1	3.24		

위 실험은 n_reno3, n_reno4 의 알고리즘을 합치면 더 좋은 성과가 나올 것이라 예측해 만든 알고리즘입니다. Cwnd 가 10000 으로 설정, 1/4 씩 감소, threshold 보다 작을 경우 3/2 배 빠르게 증가를 모두 적용한 알고리즘입니다. 아쉽게도 성능은 n_reno3 보다는 좋았지만 n_reno4 보다는 좋지 않았습니다.

이미 cwnd 가 10000 으로 설정된 상황에서 slow start 에서 3/2 배 빠르게 증가는 지나치게 cwnd 를 증가시켜 오히려 성능을 저하시키는 것으로 보입니다.

다음은 실험은 새로운 시도를 해보았습니다. 기존의 new_reno 의 틀에서 slow_start 함수와 congestion_avoid 함수를 좀 더 세분하게 나눈 알고리즘을 적용했습니다.

앞서 초반에 window 의 증가 속도를 높이는 것이 가장 효과가 좋았던 것에 착안하여, early slow start 라는 단계를 추가했습니다. Slow start 보다 더 초기의 시점에 더 많이 window 를 증가시킵니다. Ack 의 2 배만큼 증가시켜 기존의 2 배의 window 가 rtt 마다 증가합니다.

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
    struct tcp_sock *tp = tcp_sk(sk);
    printk(KERN_INFO "tp->snd_cwnd is %d\n", tp->snd_cwnd);

    if (!tcp_is_cwnd_limited(sk))
        return;

    // changed early slow start phase added
    if (tp->snd_cwnd <= (tp->snd_ssthresh) / 2) {
        //changed
        /* In "early slow start", cwnd is increased by the number of
        acked = tcp_slow_start_custom(tp, acked);
        acked = acked * 2;
        if (!acked)
            return;
    }
    else if (tp->snd_cwnd <= tp->snd_ssthresh) {
        /* In "slow start", cwnd is increased by the number of ACKed
        acked = tcp_slow_start_custom(tp, acked);
        if (!acked)
            return;
    } else {
        /* In "congestion avoidance", cwnd is increased by 1 full pacl
         * per round-trip time (RTT), which is approximated here by tl
         * ACKed packets divided by the current congestion window. */
        tcp_cong_avoid_ai_custom(tp, tp->snd_cwnd, acked);
    }

    /* Ensure that cwnd does not exceed the maximum allowed value */
    tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_cwnd_clamp);
}
```

```
// changed early slow start phase added
if (tp->snd_cwnd <= (tp->snd_ssthresh) / 8) {
    //changed
    /* In "early slow start", cwnd is increased by the numb
    acked = tcp_slow_start_custom(tp, acked);
    acked = acked * 8;
    if (!acked)
        return;
}
// changed early slow start phase added
else if (tp->snd_cwnd <= (tp->snd_ssthresh) / 4) {
    //changed
    /* In "early slow start", cwnd is increased by the numb
    acked = tcp_slow_start_custom(tp, acked);
    acked = acked * 4;
    if (!acked)
        return;
}
// changed early slow start phase added
else if (tp->snd_cwnd <= (tp->snd_ssthresh) / 2) {
    //changed
    /* In "early slow start", cwnd is increased by the numb
    acked = tcp_slow_start_custom(tp, acked);
    acked = acked * 2;
    if (!acked)
        return;
}
else if (tp->snd_cwnd <= tp->snd_ssthresh) {
    /* In "slow start", cwnd is increased by the number of
    acked = tcp_slow_start_custom(tp, acked);
    if (!acked)
```

그림과 같이 slow start 를 더 세분하게 나누고 초반일수록 window 를 크게 증가시키도록 설정했습니다.

결과는 예상외로 좋지 않았습니다.

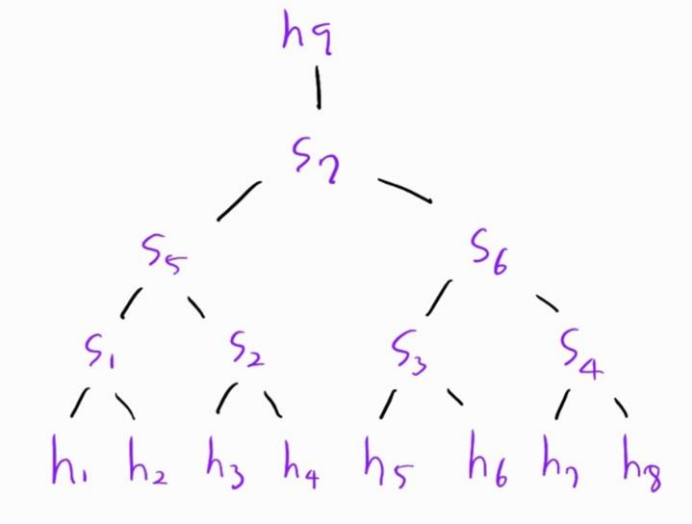
a_reno0		early slow start 추가	
bw	rtt		
26.1	3.69	bw	23.8
24.5	3.54	rtt	3.32
24.8	2.84	bw-sd	1.462945
24.3	4.19	rtt-sd	0.396882
20.9	2.73	bw-sd	0.809835
24.6	3.31		1.658061
23.7	3.33		1.920937
24.1	2.76		
		rtt-sd	0.557524
21.3	3.64		0.332303
21.9	3.57		0.300818
23.9	3.25		
25.5	2.99		

a_reno01		early slow start 8개 추가	
bw	rtt		
22.2	2.88	bw	23.75
22.2	3.32	rtt	3.568333
23.9	3.26	bw-sd	1.458406
23.7	2.65	rtt-sd	0.896652
24.9	2.57	bw-sd	0.927362
20.9	3.53		2.132878
25	3.4		1.314978
25.5	3.75		
		rtt-sd	0.318264
23.9	2.72		0.515647
23.2	6.98		1.856044
26.1	3.48		
23.5	4.28		

a_reno10	early slow start 16개 추가		
bw	rtt	decrease 단계 추가	
20.8	2.64	bw	23.38333
22.9	2.88	rtt	3.53
23.7	3	bw-sd	1.352257
22.2	3.11	rtt-sd	0.787332
20.6	5.95	bw-sd	1.230176
24	5.7		1.65328
22.8	3.1		1.173314
24.2	3.08		
		rtt-sd	0.201556
23.4	2.65		1.582369
25.7	2.85		0.578072
24.4	3.5		
25.9	3.9		

단계를 추가하면 할수록 성능이 조금씩 악화되었습니다. 의외로 단순하게 slow start 단계에서 ack 에 3/2 를 적용하는 것이 더 좋은 성능을 보이고 있습니다.

이제는 다른 토폴로지를 테스트 해보기로 했습니다. 다양한 토폴로지에 실험을 해봐야 이게 특정 형태의 네트워크에 특화된 알고리즘인지 알 수 있습니다.



위의 그림과 같은 토폴로지를 사용했습니다. 이전과 비슷하게 h1 h9, h3 h9, h5 h9, h7 h9 사이의 bw, rtt, 표준편차들을 구했습니다. 실험 방식도 동일합니다. 기준이 될 reno_custom.c 의 성능을 측정했습니다.

reno default				reno custom default			
bw	rtt			bw	rtt		
22.7	4.33	bw	22.85	1.63	5.48	bw	18.7775
23.6	5.45	rtt	5.1525	16.6	6.75	rtt	5.295833
22.5	3.16	bw-sd	1.171689	21.3	3.89	bw-sd	4.389696
21.4	5.63	rtt-sd	1.73509	22.2	5.57	rtt-sd	1.540499
22.5	4.66	bw-sd	0.903696	19.5	5.31	bw-sd	9.523632
20.6	5.9		1.273774	21.6	6.32		0.960469
23.3	3.57		1.337597	21.3	2.84		2.684989
23.3	9.67			21.3	5.5		
		rtt-sd	1.143485			rtt-sd	1.174149
21.6	5.01		2.656401	16	7.2		1.50043
23.9	5.94		1.405383	20.9	6.68		1.946919
24.4	2.84			21.1	2.86		
24.4	5.67			21.9	5.15		

전반적인 기조는 작은 토폴로지와 비슷했습니다. 더 많은 네트워크가 한곳에 몰리다보니 성능이 떨어졌습니다.

저번 실험에서 성능이 좋았던 new_reno3, new_reno4 알고리즘을 적용해 보았습니다.

n_reno3				n_reno4			
bw	rtt			bw	rtt		
20.6	5.98	bw	20.98333	20.2	7.17	bw	20.3
22.2	6.45	rtt	5.3975	18.5	6.12	rtt	5.735833
19.8	2.96	bw-sd	1.181693	19.5	3.18	bw-sd	1.412347
22.6	6.44	rtt-sd	1.668434	20.2	4.38	rtt-sd	1.643211
20.3	6.1	bw-sd	1.321615	19.1	7.18	bw-sd	0.804156
22	8.16		1.030776	21.1	6.47		1.219289
19.9	3.75		1.192686	22	4.45		2.213594
21.7	5.01			21	6.6		
		rtt-sd	1.679372			rtt-sd	1.77759
19	5.5		1.868877	17.2	5.75		1.1907
21.1	5.59		1.457052	21.6	8.34		1.961343
20.8	2.82			21.9	3.55		
21.8	6.01			21.3	5.64		

두 알고리즘 모두 reno_custom 보다는 더 성능이 좋은 것으로 나타났습니다. 하지만

이번에는 n_reno3 이 성능이 근소한 차이로 더 좋은 것으로 나왔습니다. Link utilization, rtt, bw fairness 는 n_reno3 이 앞서고 rtt fairness 는 거의 동일한 것으로 보입니다. 원인을 분석하는 과정 중 재미있는 사실을 발견했는데,

```
[ 5067.424799] tp->snd_cwnd is 12556
[ 5067.434993] tp->snd_cwnd is 12556
[ 5067.435053] tp->snd_cwnd is 12556
[ 5067.435058] tp->snd_cwnd is 12556
[ 5067.446255] tp->snd_cwnd is 12556
[ 5067.447056] tp->snd_cwnd is 12556
[ 5067.447295] tp->snd_cwnd is 12556
[ 5067.447888] tp->snd_cwnd is 12556
[ 5067.447907] tp->snd_cwnd is 12556
[ 5067.448136] tp->snd_cwnd is 12556
[ 5067.448145] tp->snd_cwnd is 12556
[ 5067.477285] tp->snd_cwnd is 12556
```

바로 윈도우의 크기가 감소하는 경향을 보였다는 것입니다.

New_reno3 알고리즘이 window 사이즈를 10000 으로 설정하고 시작하므로 이상적인 window 사이즈에 더 가깝게 시작하게 되는 것입니다. 이런 경향이 이번 토폴로지에서는 new_reno3 이 더 좋은 결과를 낸 원인인 것 같습니다.

결론적으로 두 가지 종류의 토폴로지에서 가장 좋은 결과를 낸 알고리즘은 New_reno3 인 것으로 나왔습니다.

개인적으로 세부적으로 단계를 나눈 a_reno 알고리즘이 더 성능이 좋을 것이라 예측했습니다. 초반일 수록 Congestion 이 적을거라 더 많은 window 를 증가시키는게 이상적이라 생각했습니다. 그래서 더 많은 단계를 세부적으로 설정했습니다. 하지만 의외로 단순히 윈도우의 크기를 3/2 배 한 것이 효율적으로 나와 흥미로웠습니다.

개선될 수 있는 점은 토폴로지가 너무 집중화되어서 high congestion 상황에서만 적용되는 알고리즘일 수도 있다는 것과, window 사이즈를 10000 으로 고정한 알고리즘이 너무 특수한 상황에만 유효한 방식이 아닌가 생각합니다. 두 경우의 토폴로지에서도 유효했지만 설계한 토폴로지가 의도적으로 congestion 이 많은 상황입니다. 적은 데이터를 주고 받는 일반적인 상황이라면 효율이 다르게 나올 수도 있다고 생각했습니다.

감소폭을 절반에서 1/4 로 바꾸는 것과 window 를 특정값으로 고정하는 것이 알고리즘의 유연성에 좋을 것 같지 않을 것 같습니다. 이를 해결할 수 있는 방법으로는 네트워크의 상황에 대한 정보를 받아드리고 10000 이란 고정값을 유동적으로 변화시킬 수 있는 방법을 찾으면 더 좋은 성과를 낼 수 있다고 생각합니다.

읽어주셔서 감사합니다.