

컴퓨터 네트워크 개론 과제
Application Level Congestion Control and Reliable Delivery
염익준 교수님

2018310520
김세란

목차

1. 코드 설명

- 0) 실험 조건에 대한 변수와 실험 중 측정값에 대한 변수
- 1) queue 를 이용한 loss emulation 코드(server.py 스레드 구조)
- 2) 3 dup ack 과 time out 에 의한 loss detection 및 retransmission 코드(client.py 스레드 구조)
- 4) loss-based congestion control 코드
- 5) delay-based congestion control 코드

2. 성능 평가

- 0) queue 크기와 bandwidth 에 따른 성능 평가
- 1) window 크기가 고정된 경우와 loss-based congestion control 비교
- 2) loss-based congestion control 과 delay-based congestion control 비교

1. 코드 설명

0) 실험 조건 에 대한 변수와 실험 중 측정값에 대한 변수

client.py		server.py	
조건값		조건값	
cc_mode	congestion control(cc) 모드 initial 이면 wnd 고정 loss 이면 loss-based cc delay 이면 delay-based cc	queue_size	큐의 크기(단위는 패킷 개수)
win	초기 윈도우 크기	bandwidth	링크(큐)에서 나가는 속도 (단위는 패킷 개수 / 초)
ssh	초기 스레시홀드 값 loss-based cc 일때 사용	no_pkt	보내는 패킷의 개수
no_pkt	보내는 패킷의 개수	측정값	
# of loss detection	timeout 에 의한 loss detection(retransmission)횟수와 3dup 에 의한 loss detection(retransmission)횟수를 구분하여 측정	# of packet loss	실제 buffer overflow 에 의해 발생한 패킷 loss 의 개수
packet 별 rtt	ack 를 받은 시각 - packet 을 보낸 시각	기타 조건: queue 의 drop policy 는 tail drop 으로 설정함. 즉 큐가 꽉 찬 상태일 경우 마지막으로 온 패킷은 큐에 들어가지지 못하 버려짐(loss 발생) queue 의 scheduling 기법으로는 FCFS 으로 설정한다. 즉 먼저 들어온 패킷을 먼저 내보낸다.	
시간당 보낸 패킷 개수	시간당 재전송을 포함한 전체 보낸 패킷 개수		

1) loss emulation (server.py 스레드 구조)

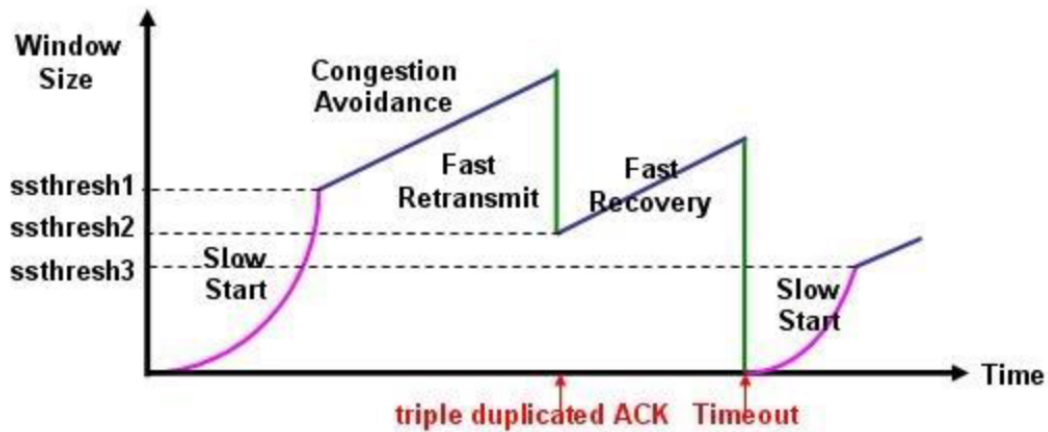
server.py 총 3 개의 스레드로 이루어져 있다. 이는 각 작업 간의 시간 동기화에 의한 오차를 줄이기 위한 설계이다.		
thread1: enqueueing	thread2: dequeing	thread3: main
<p>링크에 패킷이 들어오는 부분이다.</p> <p>1) 서버소켓으로 부터 메세지 수신 2) 버퍼(pkt_queue)의 크기(queue_size)를 확인하고, 버퍼에 패킷을 추가한다. 버퍼 오버플로우가 발생할 경우 패킷을 drop 한다. 이때 drop policy 로는 tail drop 을 사용한다. (1-0 에서 설명)</p>	<p>링크에서 패킷이 나가는 (즉 서버에 패킷이 들어오는) 부분이다.</p> <p>큐의 크기에 따라 시간당 링크에서 나가는 패킷의 개수가 달라진다.</p> <p>1) 링크의 대역폭(bandwidth)에 맞게 1/bandwidth 의 시간 간격으로 버퍼에서 패킷을 꺼낸다. 이때 스케줄링 기법은 FCFS 로 먼저 들어온 패킷을 먼저 내보낸다. 2) 이때 꺼낸 패킷은 dequeued_pkt 이라는 리스트에 저장한다. 이는 main 스레드의 server 가 수신한(링크로 부터 받은) 패킷들 이다.</p> <p>이렇게 하여 main 스레드에서 발생할 수 있는 처리 시간과 독립적으로 bandwidth 에 맞게 dequeue 를 한다.</p>	<p>링크에서 패킷을 받은 서버에 해당하는 부분이다.</p> <p>1) dequeued_pkt 에 접근하여 링크로 부터 받은 패킷을 순서대로 읽는다.</p> <p>2) 패킷의 sequence number 가 rcv_base(cum ACK)라면 rcv_base 를 증가시키고, ACK 을 client 에게 전송한다.</p>

2) loss detection by 3 dup ack and time out (client.py 스레드 구조)

client.py 총 2 개의 스레드로 이루어져 있다.	
thread1: handling_ack	thread2: main
<p>클라이언트가 서버로 부터 ACK 을 받는 부분이다.</p> <p>또한 time out 이나 3 dup ack 으로 loss detection 을 수행한다.</p> <p>1) ACK 를 받으면 - estimated rtt 와 dev rtt 를 이용하여 timeout_interval 을 계산한다. - 중복 ack 와 그 개수를 계산한다. - 제대로 된 ack 를 수신한 경우 send_base 를 증가시킨다. - 설정한 congestion control 의 규칙에 따라 window 크기를 조절한다. (1-3, 1-4 에서 설명)</p> <p>2) timeout 과 3 dup ack detection 수행 timeout_flag 와 tdup_flag 를 loss detection 을 하여 main 스레드에서 재전송을 할 수 있도록 한다.</p>	<p>클라이언트가 서버에게 패킷을 보내는 부분이다.</p> <p>윈도우 크기에 따라 시간당 보내는 패킷의 개수가 달라진다.</p> <p>1) window 크기와 send_base 를 통해 보낼 수 있는 만큼 서버로 패킷을 전송한다.</p> <p>2) loss detection 이 된 경우 재전송을 한다.</p>

3) loss-based congestion control

다음은 slow start + congest avoidence + fast recovery 를 이용해 window size 를 조절하여 loss-based congestion control 을 구현한 것에 대한 설명이다. 이는 client.py 의 handlin_ack 스레드에 구현되어 있다.



window 의 크기 변화에 대한 그림

client.py – handling_ack thread

`cc_state`(congestion control state) 변수를 통해 slow start('ss') / fast recovery('fr') / congestion avoidance('ca')로 window 증가 방식을 구별할 수 있도록 했다.

window 증가시키는 경우

ack 를 받고 `send_base` 를 증가시킬 때 window 크기를 증가시킨다.

`cc_state` 에 따라

'ss'인 경우 +1 씩 증가, 나머지는 x2 씩 증가한다.

`cc_state` 는 아래 함수에서 변경된다.

window 를 감소시키는 경우

window 크기가 증가했는데, `ssh`(스레시홀드)값 보다 커진 경우, 3dup ack 으로 인해 loss detection 이 된 경우, time out 으로 인해 loss detection 이 된 경우 window 크기가 감소한다.

loss detection 의 경우 `ssh` 의 크기도 변화한다.

각 세 가지 경우는 아래 *함수에 구현된다.

*사용한 함수

ssh_check window 가 증가할때 호출 <code>cc_state</code> 가 'ca'로 변경 win 크기: 1(<code>ssh < win</code> 인 경우)	slow_start timeout 발생 시 호출 <code>cc_state</code> 가 'ss'로 변경 win 크기: 1 ssh 크기: 기존 win 크기 / 2	fast_recovery 3dup ack 감지시 호출 <code>cc_state</code> 가 'fr'로 변경 win 크기: 1/2 배 ssh 크기: 기존 win 크기 / 2
--	--	---

4) delay-based congestion control

$T_m(\text{measured throughput}) = (\text{지난 RTT 에서 보낸 데이터}) / \text{RTT}_m(\text{RTT measured})$ 가

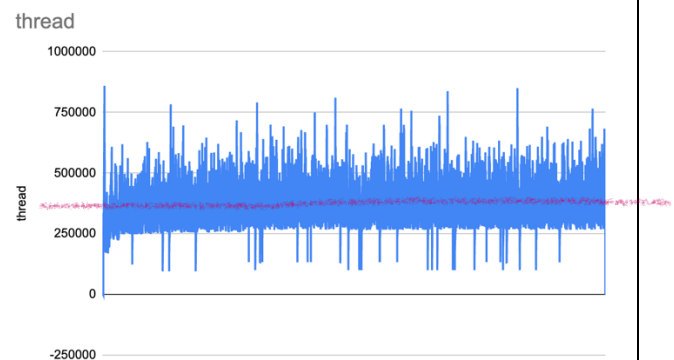
$T_{uc}(\text{uncongested throughput}) = (\text{현재 window size}) / \text{RTT}_{min}(\text{uncongested path 의 delay})$ 와 비슷한 경우 window 크기를 선형적으로 증가(+1), 그렇지 않고 $T_m \ll T_{uc}$ 일 경우 window 크기를 선형적으로 감소(-1) 시킨다.

구현 시에 T_m 과 T_{uc} 가 얼마나 가깝냐 정하는 기준은,

loss-based congestion control 을 할 $T_m - T_{uc}$ 의 변동을 보고 그 중심 값으로 임의로 설정하였다.

예를 들어
loss-based congestion control 을
할 때
시간에 따른 $T_m - T_{uc}$ 변화가 다음과
같다면, 350000 으로 설정

→ 이경우 그 이상의 값으로 설정할
때 보다 delay-based 의 성능이
나아지는 것을 볼 수 있었음.



2. 성능 평가:

0) queue 크기와 bandwidth 에 따른 성능 평가

- 실험 결과:

	기본	bandwidth줄임	bandwidth늘림	q_size늘림	q_size줄임
numofpacket	1000	1000	1000	1000	1000
winsize	10	10	10	10	10
link bandwidth	800	500	1000	800	800
queue_size	50	50	50	35	75
# of sendpacket / 1 sec (include retransmit)	355.92936547 73385	339.20988257 154625	359.219101431 8502	334.26543703 25322	361.154510574 23644
# fo sendpacket /sec (not include retransmit)	14.7817336881 65557	12.7484171140 83969	14.6482527191 55495	16.830242033 761248	12.002476257 036772
# of pkt loss detection (3dup/t.o)	2316/2308 14752	2567/2573 16394	2363/2366 14932	1887/1891 12177	2912/2923 18653
# of actual pkt loss					

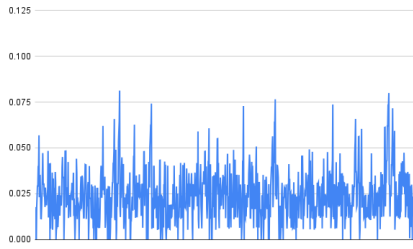
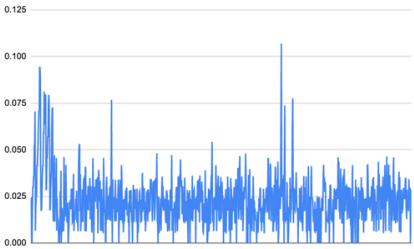
- 결과 분석:

해당 결과에서 보았을 때 대역폭이 1000 일때 대역폭이 500 일 때에 비해 패킷 손실이 감소했다. 패킷 손실은 큐에 패킷이 들어오는 속도에 비해 나가는 속도가 빠를 수록 적게 일어나므로 들어오는 속도 / 나가는 속도 즉 $359/1000 < 339/500$ 로 이를 설명할 수 있다. 또한 큐의 크기가 감소하였을 때 패킷 손실이 증가하는 것을 확인할 수 있다.

2-1),2-2) 의 경우 다음 과 같은 조건을 동일하게 하여 3 가지 실험을 진행하였다.
패킷 개수 총 1000 개, 초기 윈도우 크기 10 개, 큐 크기 50, 대역폭 800

1) window 크기가 고정된 경우와 loss-based congestion control 비교

- 실험 결과:

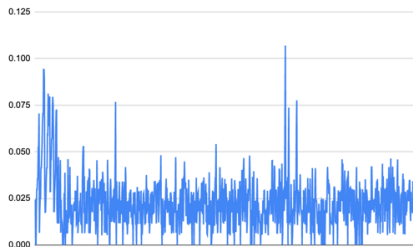
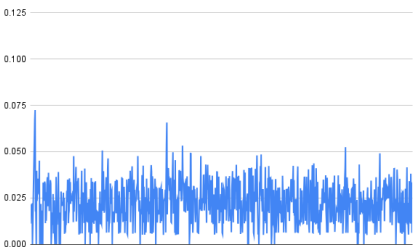
	window 크기가 고정된 경우	loss-based congestion control
rtt 분포	 <p>평균: 0.02280748677 중앙값: 0.02256715298</p>	 <p>평균: 0.0228078677 중앙값: 0.2256715298</p>
loss detection 횟수(t.o/3dup)/ 실제 loss 횟수	2484(7 / 2477)/ 12389	2201(10/2191)/ 9888
초당 보낸 패킷 수 재전송제외/전체	13.886547685699762 /356.8981620701696	15.524024010319522 /285.42470545373476

- 결과 분석:

loss-based congestion 에서 loss 를 감지할 때 window 크기(전송 속도 de)를 조절하여 혼잡도를 줄이고 실제 loss 가 12389 에서 9888 로 감소하였다. 불필요한 3 번째 데이터로 보아 불필요한 재전송이 매우 감소한 것이 이에 대한 요인 중 하나라고 볼 수 있다. 평균 rtt 는 두 실험에서 동일하지만, 분포로 보았을 때 전반적으로 loss-based cc 의 상황에서 적은 rtt 를 보인다. 따라서 loss-based cc 이 window 크기가 고정된 경우보다 성능이 좋다.

2) loss-based congestion control 과 delay-based congestion control 비교

- 실험 결과:

	loss-based congestion control	delay-based congestion control
rtt 분포	 <p>평균: 0.0228078677 중앙값: 0.02256715298</p>	 <p>평균:0.02248613518 중앙값: 0.02289676666</p>
loss detection 횟수(t.o/3dup)/ 실제 loss 횟수	2201(10/2191)/ 9888	2821(10/2191)/ 4291

초당 보낸 패킷 수	15.524024010319522	12.167614560253305
재전송제외/전체	/285.42470545373476	/179.19245962885043

- 결과 분석:

delay-based cc 의 경우 loss 가 발생하기 전에 rtt 와 throughput 을 측정해 혼잡도를 예상하고 window 크기를 조절한다. 따라서 loss-based cc 에 비해 실제 loss 에 대한 횟수가 거의 절반에 가깝게 작은 것을 볼 수 있다. rtt 의 분포를 보았을 때에도 loss 로 인해 rtt 가 갑자기 증가하는 것으로 보이는 모양을 loss-based 에서는 발견할 수 있지만 delay-based 에서는 발견할 수 없다.

추가로, delay-based 에서 보다 민감하게 congestion 을 예상하도록 기준을 변경하게 되면, loss-based 에 비해 절반의 rtt 분포를 가지도록 할 수 있음을 확인하였다.

결론:

실제 패킷 손실 수와 rtt 의 중앙값으로 보았을 때,
window 크기 고정 < loss-based cc < delay-based cc 순으로 성능이 좋으며,
delay-based congestion control 을 적용하였을 때 가장 좋은 성능을 보였다.

- 감사합니다 -