REPORT

컴퓨터 네트워크 - 1차 프로젝트

광운대학교
KwangWoon University

과목명	|	컴퓨터 네트워크
담당교수	|	박재성 교수님
학과	|	정보융합학부
학년	|	2학년
학번	|	2019204054
이름	|	김현준
제출일	|	2020.11.05

Computer Network Project 01

2019204054 김현준

▼ CONTENT

Introduction
Implementation
    rdt 3.0
    go-back-N
    selective-repeat
    Timeout
    Loss
    Efficiency
Performance Test & Consideration
    rdt 3.0
    go-back-N
        WindowSize = 10 이고 RTT가 다를 때
        WindowSize = 100 이고 RTT가 다를 때
        RTT = [8ms, 12ms] 이고 WindowSize가 다를 때
        RTT = [80ms, 120ms] 이고 WindowSize가 다를 때
    selective-repeat
        WindowSize = 10 이고 RTT가 다를 때
        WindowSize = 100 이고 RTT가 다를 때
        RTT = [8ms, 12ms] 이고 WindowSize가 다를 때
        RTT = [80ms, 120ms] 이고 WindowSize가 다를 때
    Conclusion
Reference

Introduction

본 프로젝트는 수업 시간에 배운 **rdt 3.0**, **Go-Back-N**, **Selective-Repeat** 프로토콜을 각각 구현하고, 네트워크 환경 변수에 따른 효율 변화의 측정을 목표로 한다.

*sender.py*는 Client의 역할을, *receiver.py*는 Server의 역할을 한다. 두 프로그램 모두 sys모듈을 통해 명령행 인자를 받아 프로토콜을 결정하며, 항상 두 프로그램의 프로토콜 인자는 같다고 가정한다. 적절하지 않은 인자가 들어오면 즉시 프로그램을 종료한다.

*sender.py*는 `rdt3_send()`, `gbn_send()`, `sr_send()` 함수를 가지며, 각각 `ack_receive()`, `ack_receive()`, `sr_ark_receive()` 함수를 통하여 ACK를 받는다. 각 함수는 socket 객체

를 인자로 받는다. LOSS_PROB(손실 확률)을 제외한 모든 환경 변수는 *sender.py*에 위치한다.

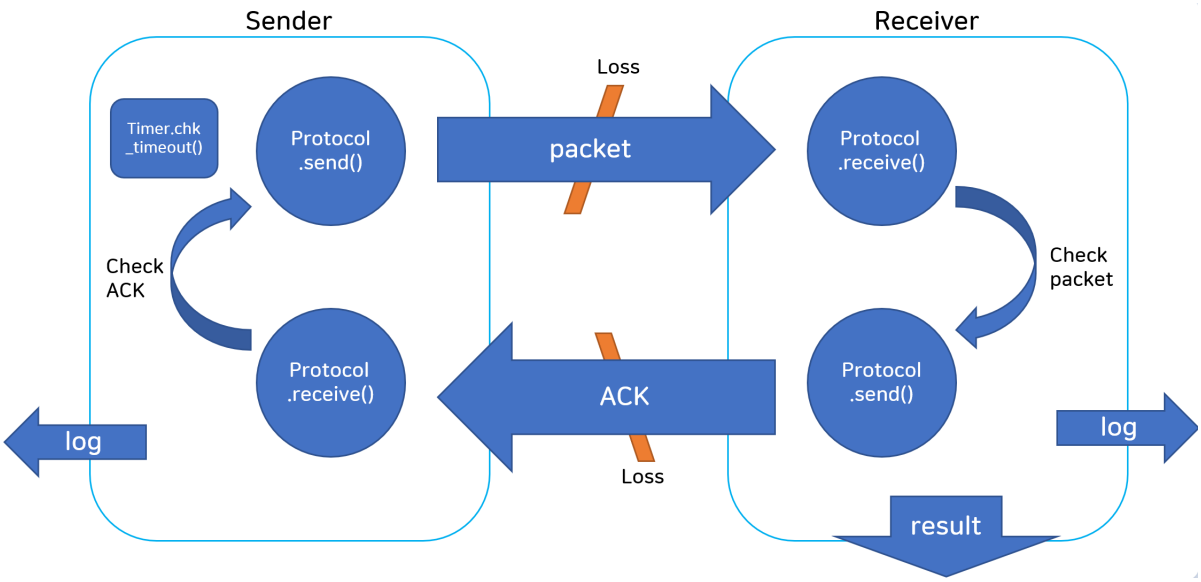
*receiver.py*는 `rdt3_receive()`, `gbn_receive()`, `sr_receive()` 함수를 가지며, 각각 패킷을 받고, ACK를 보내는 과정까지 구현되어 있다. 각 함수는 socket 객체를 인자로 받는다.

*util.py*는 각각의 프로그램에서 쓰이는 함수들과 타이머 클래스가 정의되어 있다. 확률적으로 LOSS를 발생시키는 `send()`함수와, 패킷을 받는 `recv()`함수, 패킷을 만드는 `make_packet()`함수, 패킷을 해체하는 `extract_packet()`함수로 이루어진다. 단, 패킷은 시퀀스 넘버로만 이루어져 있다고 가정한다.

콘솔 두 개를 열고 각각 `python receiver.py <protocol>` , `python sender.py <protocol>`을 입력 함으로써 시뮬레이션 할 수 있다. *sender.py*에 있는 MAXIMUM_TIME 변수값만큼만 시뮬레이션을 진행하며, 시간이 종료되면 즉시 시뮬레이션을 종료한다. 전송 시뮬레이션이 끝나면 *recvlog.txt*와 *sendlog.txt*에 각 프로그램의 로그가 작성되며, 시뮬레이션에 사용되었던 환경 변수들과 전송된 데이터의 개수를 "프로토콜 이름".txt에 저장한다. 모든 파일은 *append*모드로 연다.

*graph.py*는 손실 확률에 따른 전송 효율의 그래프를 얻어내기 위한 함수로, sys모듈을 통해 명령행 인자를 받아 데이터 파일을 결정하며, 각 프로토콜 데이터 파일은 항상 LOSS확률을 기준으로 내림차순 정렬된 데이터를 가진다고 가정한다.

그래프에는 matplotlib 모듈, 확률과 무작위 값 추출에는 random모듈, 인자를 위한 sys 모듈 통신을 위한 socket모듈, 시간 측정과 로그 작성을 위한 time, datetime 모듈을 사용하며, timeout 구현을 위해 병렬 쓰레드를 사용하는데, 이를 위해 _thread 라는 저수준 모듈을 사용하였다.



간략한 흐름도

Implementation

rdt 3.0

- rdt 3.0의 구현에 사용된 알고리즘은 다음과 같다.

1. sender가 현재 시퀀스 넘버에 해당하는 패킷을 만들어 보낸다.
2. *timer*가 멈춰있는 상태라면 시작하고, $RTT(\tau)$에 해당하는 시간 만큼 *sleep*한다.
3. receiver가 패킷을 받는다. (data loss가 발생하면 4번 과정을 건너뛴다.)
4. 받은 패킷을 해체하여 시퀀스 넘버에 따른 적절한 행동을 취한다.
    1. 시퀀스 넘버가 -1인 패킷을 받았다면, 프로그램을 종료한다.
    2. 옳은 차례의 시퀀스 넘버를 가진 패킷을 받았다면 저장하고, 받은 시퀀스에 대한 패킷을 만들어서 보낸다.
    3. 그른 차례의 시퀀스 넘버를 가진 패킷(중복 패킷)을 받았다면, 받은 시퀀스에 대한 패킷을 만들어서 보낸다.
5. sender가 ACK를 받는다. (ACK loss가 발생하면 6번 과정을 건너뛴다.)
6. 받은 ACK를 해체하여 ACK에 따른 적절한 행동을 취한다.
    1. 적절한 ACK를 받았다면, 보내야 할 다음 시퀀스 넘버를 업데이트한다.
    2. 적절하지 않은 ACK를 받았다면, 아무 행동도 하지 않는다.
    3. *timer*를 리셋한다.
7. 6번 과정 후에 sender는 다음에 보내야 할 시퀀스 넘버를 판단한다.
    1. Loss가 발생하였다면 *timer class*가 가진 메소드가 timeout을 판별한다. 현재 시퀀스 넘버를 업데이트하지 않고, 1번 과정으로 돌아간다.
    2. Loss가 발생하지 않았다면, 현재 시퀀스 넘버를 다음 시퀀스 넘버로 업데이트하고, 1번 과정으로 돌아간다.

- 위 과정을 *sender.py*에 있는 MAXIMUM_TIME이 될 때까지 반복한다.
- MAXIMUM_TIME이 되었다면 -1이 담긴 패킷을 receiver에 보낸다. 그때, receiver는 위의 4-1 과정을 수행하게 된다.
- 시퀀스 넘버가 0부터 차례대로 증가하게 되어있다. 이는 로그를 편하게 보기 위해 임의로 수정한 것으로, 본래 rdt 3.0 프로토콜의 의도대로 돌아가려면 *sender.py*의 57,

58번째 줄을 서로 바꾸고, *receiver.py*의 48, 49번째 줄을 서로 바꾸면 된다.

go-back-N

- go-back-N의 구현에 사용된 알고리즘은 다음과 같다.

1. sender가 base부터 시작하여 현재 시퀀스 넘버에 해당하는 패킷을 만들어 보낸다. WINDOW_SIZE만큼 반복한다.
2. *timer*가 멈춰있는 상태라면 시작하고, $RTT(\tau)$에 해당하는 시간 만큼 *sleep*한다.
3. receiver가 패킷을 받는다. (data loss가 발생하면 4번 과정을 건너뛴다.)
4. 받은 패킷을 해체하여 시퀀스 넘버에 따른 적절한 행동을 취한다.
    1. 시퀀스 넘버가 -1인 패킷을 받았다면, 프로그램을 종료한다.
    2. 옳은 차례의 시퀀스 넘버를 가진 패킷을 받았다면 저장하고, 받은 시퀀스에 대한 ACK 패킷을 만들어서 보낸다.
    3. 그른 차례의 시퀀스 넘버를 가진 패킷(중복 패킷)을 받았다면, 받은 시퀀스에 대한 ACK 패킷을 만들어서 보낸다.
5. sender가 ACK를 받는다. (ACK loss가 발생하면 6번 과정을 건너뛴다.)
6. 받은 ACK를 해체하여 ACK에 따른 적절한 행동을 취한다.
    1. 적절한 ACK를 받았다면, *base*를 업데이트(Window를 Shift 하는 효과가 있다.)한다.
    2. 적절하지 않은 ACK를 받았다면, 아무 행동도 하지 않는다.
    3. *timer*를 리셋한다.
7. 6번 과정 후에 sender는 다음에 보내야 할 시퀀스 넘버를 판단한다.
    1. Loss가 발생하였다면 *timer class*가 가진 메소드가 *timeout*을 판별한다. *timeout*이 발견되면 *base*를 timeout이 생긴 시점으로 이동시키고 1번 과정으로 돌아간다.
    2. Loss가 발생하지 않았다면, *base*는 WINDOW_SIZE가 된다. 1번 과정으로 돌아간다.

- 위 과정을 sender.py에 있는 MAXIMUM_TIME이 될 때까지 반복한다.
- MAXIMUM_TIME이 되었다면 -1이 담긴 패킷을 receiver에 보낸다. 그때, receiver는 위의 4-1 과정을 수행하게 된다.

selective-repeat

- selective-repeat의 구현에 사용된 알고리즘은 다음과 같다.
- 프로그램의 기능을 구현하기 위하여 *acked*라는 전역 배열을 사용한다.
- 이는 Bool 대수를 저장하는 배열로, 초기에 WINDOW_SIZE만큼 선언되어 있다.
- 시퀀스 넘버를 인덱스로 사용한다. 해당 시퀀스 넘버에 대한 ACK를 받았을 때, 값을 True로 바꾼다.

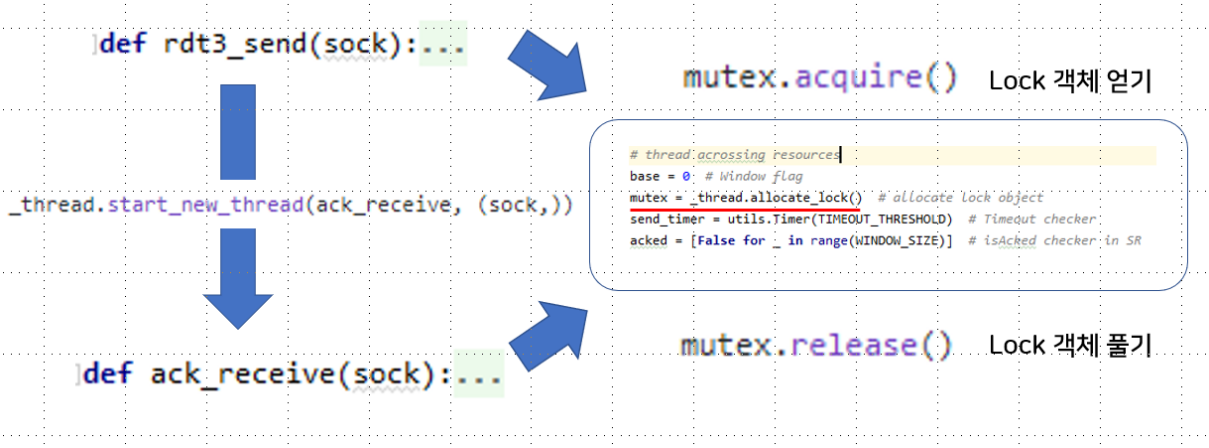
1. sender가 *base*부터 시작하여 현재 시퀀스 넘버에 해당하는 패킷을 만들어 보낸다. 배열의 길이만큼 반복한다.
    - 만약 현재 시퀀스 넘버에 해당하는 acked[] 값이 True일 경우, 패킷 전송 과정 자체를 건너뛴다.
2. *timer*가 멈춰있는 상태라면 시작하고, RTT(t)에 해당하는 시간 만큼 *sleep*한다.
3. receiver가 패킷을 받는다. (data loss가 발생하면 4번 과정을 건너뛴다.)
4. 받은 패킷을 해체하여 시퀀스 넘버에 따른 적절한 행동을 취한다.
    1. 시퀀스 넘버가 -1인 패킷을 받았다면, 프로그램을 종료한다.
    2. 옳은 차례의 시퀀스 넘버를 가진 패킷을 받았다면 저장하고, 받은 시퀀스에 대한 ACK 패킷을 만들어서 보낸다. 값을 저장할 때 적절한 순서가 아니라면 정렬한다.
    3. 그른 차례의 시퀀스 넘버를 가진 패킷(중복 패킷)을 받았다면 저장하고, 받은 시퀀스에 대한 ACK 패킷을 만들어서 보낸다.
5. sender가 ACK를 받는다. (ACK loss가 발생하면 6번 과정을 건너뛴다.)
6. 받은 ACK를 해체하여 ACK에 따른 적절한 행동을 취한다.
    1. 받은 ACK에 대한 *acked*[] 값을 True로 바꾼다.
    2. *acked*[*base*]값이 *True*이면, Window를 Shift하고 *acked*[] 의 길이를 1 늘린다.
    3. *timer*를 리셋한다.
    4. *acked*[]를 순회하여 제일 처음 만나는 *False* 의 인덱스값을 *base*로 가진다.
7. 6번 과정 후에 sender는 다음에 보내야 할 시퀀스 넘버를 판단한다.

1. 모종의 이유(Loss, Timeout)로 받지 못한 ACK는 acked[]가 False인 인덱스를 찾음으로써 추출할 수 있다.
2. *base*값을 추출된 인덱스로 바꾼다. (6-4에 대한 검증)
3. 1번 과정으로 돌아간다.

- 위 과정을 *sender.py*에 있는 MAXIMUM_TIME이 될 때까지 반복한다.
- MAXIMUM_TIME이 되었다면 -1이 담긴 패킷을 receiver에 보낸다. 그때, receiver는 위의 4-1 과정을 수행하게 된다.

Timeout

- Timer 클래스를 통해 구현한다.
- 생성 인자를 통해 임계 값을 설정한다.
- `start()` 메소드가 실행되면 time 모듈의 `time()` 함수를 통하여 현재 시각을 저장한다.
- `chk_timeout()` 메소드가 실행되면 (현재 시간 - 저장된 시간)이 임계 값을 넘는지 검사한다. 넘게 되면 *timeout*으로 간주한다.
- `reset()` 메소드가 실행되면 저장된 시간을 초기화한다.
- sender가 패킷을 보낼 때 시작해야 하고, ACK를 받을 때 종료해야 하므로 *thread*를 사용한다. 록 객체를 적절히 획득/반환하여 *timeout*을 검사한다.
- 루프 백 아이피 간의 통신은 굉장히 빠르기 때문에, RTT_MIN과 RTT_MAX 사이의 난수를 *uniform* 분포로 발생 시켜 강제로 지연시킨다.
- *timeout*의 빈도를 조절하기 위해 TIMEOUT_THRESHOLD를 적절히 설정한다.



대략적인 thread 흐름도

Loss

- Loss는 utils.py의 `send()` 함수로 구현되어 있다.
- random모듈의 `random()` 함수를 통해 0과 1 사이의 난수를 가져온다.
- 가져온 난수가 설정된 LOSS_PROB을 넘어가면, Loss가 나지 않았다고 판별하고, 패킷을 정상적으로 보낸다. 이때, 함수의 반환 값은 *True*이다.
- 가져온 난수가 설정된 LOSS_PROB을 넘어가지 않으면, Loss가 났다고 판별하고, 패킷을 보내지 않는다. 이때, 함수의 반환 값은 *False*이다.
- 만약 보내야 할 패킷의 시퀀스 넘버가 -1일 경우, 프로그램의 종료를 알리는 신호이기 때문에 Loss가 나면 안 된다. 이에 대해 예외처리를 해주었다.
- sender의 `send()`에서 Loss가 발생하면 *data Loss*, receiver의 `send()`에서 Loss가 발생하면 *ACK Loss*라고 칭한다.

Efficiency

- 그래프를 그리기 위해 *matplotlib* 의 *pyplot* 모듈을 사용한다.
- `python graph.py <protocol>` 을 통해 그래프를 얻는다.
- 모든 데이터 파일은 항상 손실 확률을 기준으로 한 블록씩 내림차순으로 정렬되어 있다고 가정한다. 데이터는 정해진 순서에 따라 한 줄에 한 개씩 작성되며, 아래 사진과 같이 <변인1, 변인2> 형태로 총 18행의 데이터가 있다고 가정한다.

```
64 10 0.1 0.11 [0.08,0.12]
89 10 0.01 0.11 [0.08,0.12]
93 10 0.001 0.11 [0.08,0.12]
91 10 0.0001 0.11 [0.08,0.12]
94 10 1e-05 0.11 [0.08,0.12]
91 10 1e-06 0.11 [0.08,0.12]
94 10 1e-07 0.11 [0.08,0.12]
94 10 1e-08 0.11 [0.08,0.12]
91 10 1e-09 0.11 [0.08,0.12]
72 100 0.1 1.1 [0.8,1.2]
96 100 0.01 1.1 [0.8,1.2]
99 100 0.001 1.1 [0.8,1.2]
99 100 0.0001 1.1 [0.8,1.2]
101 100 1e-05 1.1 [0.8,1.2]
99 100 1e-06 1.1 [0.8,1.2]
100 100 1e-07 1.1 [0.8,1.2]
101 100 1e-08 1.1 [0.8,1.2]
98 100 1e-09 1.1 [0.8,1.2]
```

데이터셋 예시 (RDT_3.txt)

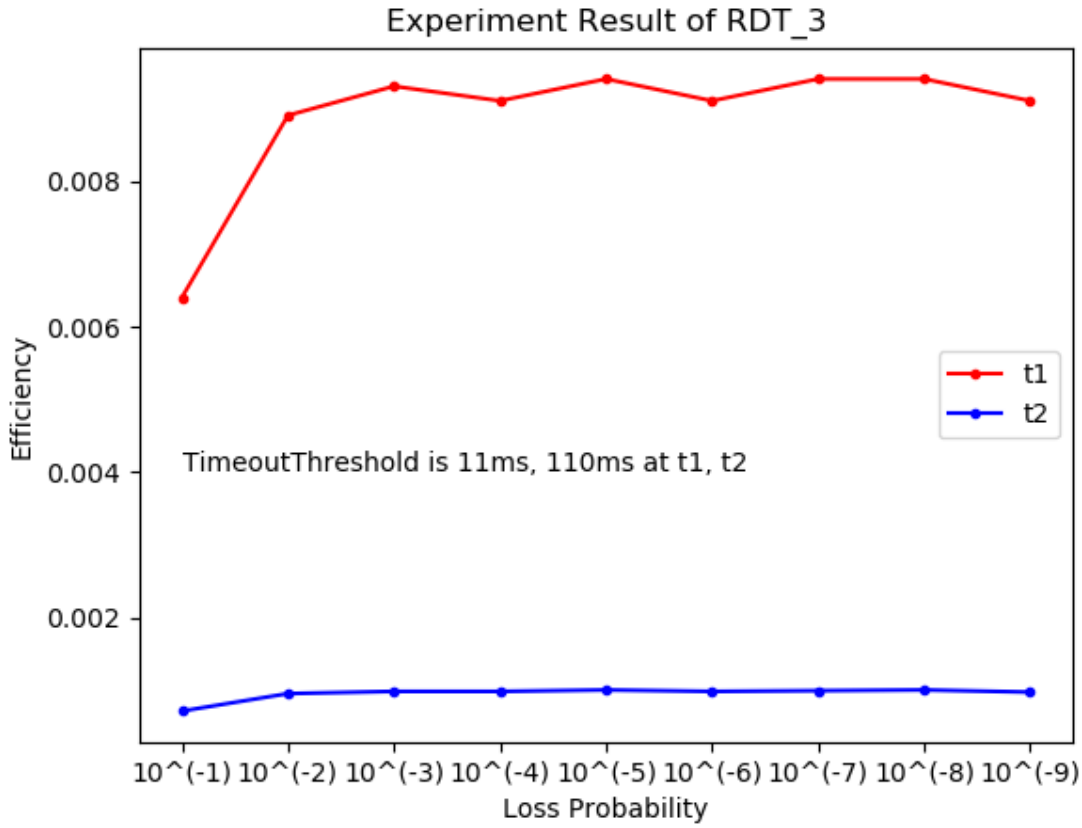
- 파일을 한 줄씩 읽어 효율 계산식 $u = n_s / t_m$ 을 통해 효율을 계산한다.
- $RTT(\tau)$나 WINDOW_SIZE를 변인으로 두고 (효율 - 손실확률) 그래프를 그린다.

Performance Test & Consideration

- 실험및 개발 환경은 다음과 같다.
  - Windows 10.0.18363 Student Ver.
  - Windows PowerShell 5.1.18362.1110
  - PyCharm 2019.3 Comunity Edition
  - Python 3.7.5
- sender와 receiver가 모두 종료될 때, receiver는 받았던 데이터를 출력한다.
- 사용되었던 프로토콜 이름으로 .txt파일을 만들고, 전송된 데이터의 개수와 사용되었던 환경 변수를 자동으로 저장한다.
- <데이터 개수> <실험 시간(초)> <손실 확률> <timeout 임계값(ms)> <$RTT(\tau)$> <Window 크기>형태로 저장되며, 각 항목은 스페이스 바를 구분자로 구분되어 있다.

- 직접 프로그램을 구동시켜 모은 데이터를 *Collected_Dataset.txt*에 저장해 두었다.
- 저장된 데이터는 다음과 같다.
  - RDT 3.0
    - 실험시간 : 10초 , timeout 임계값 : 0.11ms , τ_1일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 100초 , timeout 임계값 : 1.1ms , τ_2일때 손실 확률에 따른 전송된 데이터 개수
  - Go-Back-N
    - 실험시간 : 10초 , timeout 임계값 : 0.11ms , window 크키 : 10, τ_1일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 10초 , timeout 임계값 : 0.11ms , window 크키 : 100, τ_1일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 100초 , timeout 임계값 : 1.1ms , window 크키 : 10, τ_2일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 100초 , timeout 임계값 : 1.1ms , window 크키 : 100, τ_2일때 손실 확률에 따른 전송된 데이터 개수
  - Selective-Repeat
    - 실험시간 : 10초 , timeout 임계값 : 0.11ms , window 크키 : 10, τ_1일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 10초 , timeout 임계값 : 0.11ms , window 크키 : 100, τ_1일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 100초 , timeout 임계값 : 1.1ms , window 크키 : 10, τ_2일때 손실 확률에 따른 전송된 데이터 개수
    - 실험시간 : 100초 , timeout 임계값 : 1.1ms , window 크키 : 100, τ_2일때 손실 확률에 따른 전송된 데이터 개수

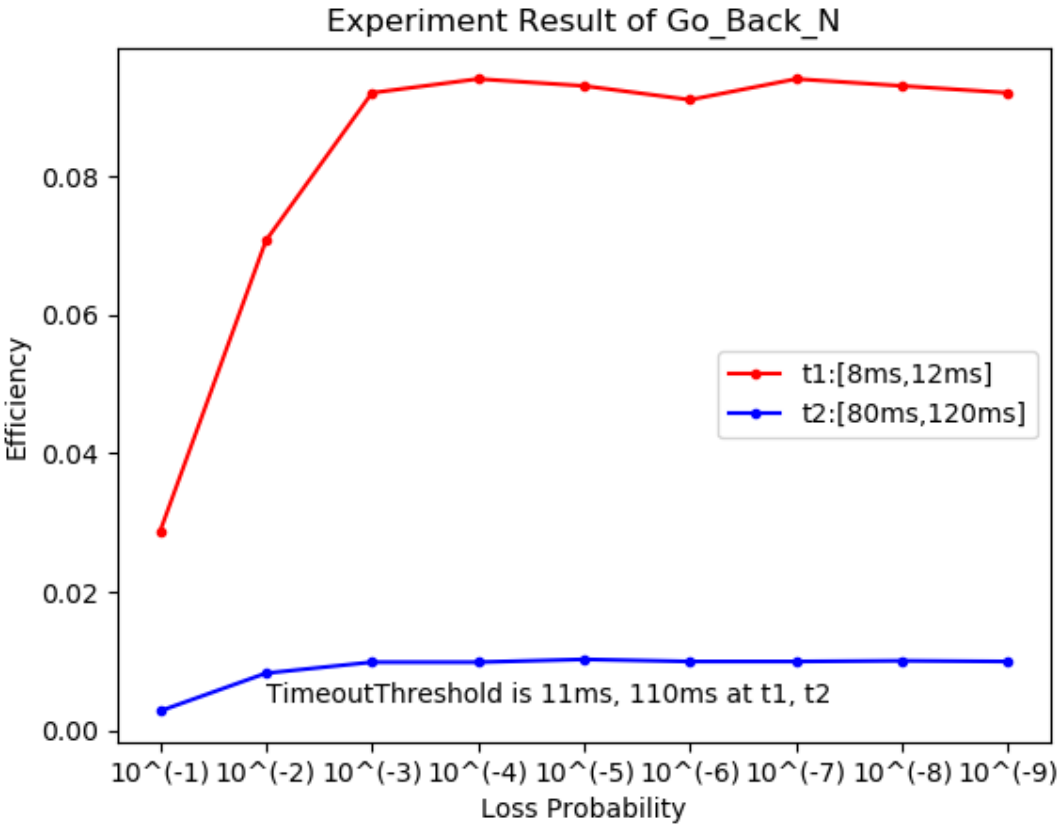
rdt 3.0



- rdt 3.0 자체가 밑의 두 프로토콜 보다 절대적인 전송 수가 적기 때문에, 효율성이 떨어진다.
- τ_1과 τ_2 간의 전송 수는 크게 차이가 나지 않지만, t_m이 10배 차이 나기 때문에 그래프가 위와 같은 양상을 보인다.
- 손실확률이 0으로 수렴하는 양상이라, 그래프도 일정 효율 값에 수렴하는 것처럼 보인다.

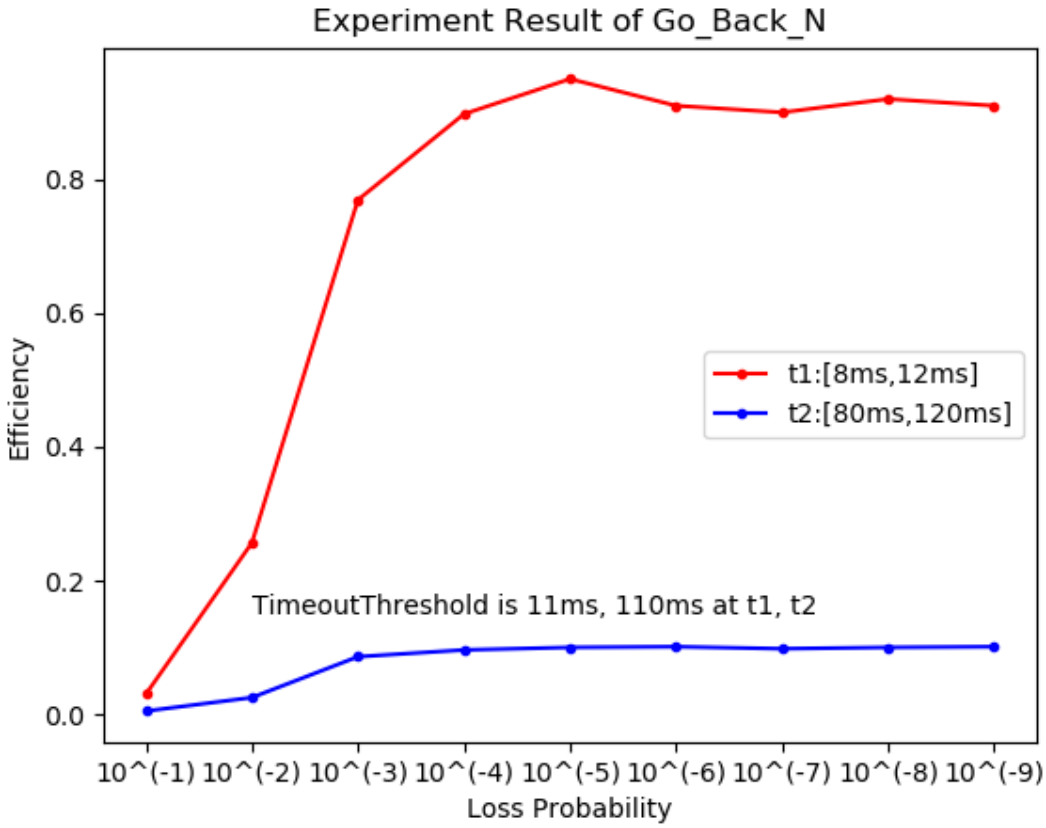
go-back-N

WindowSize = 10 이고 RTT가 다를 때



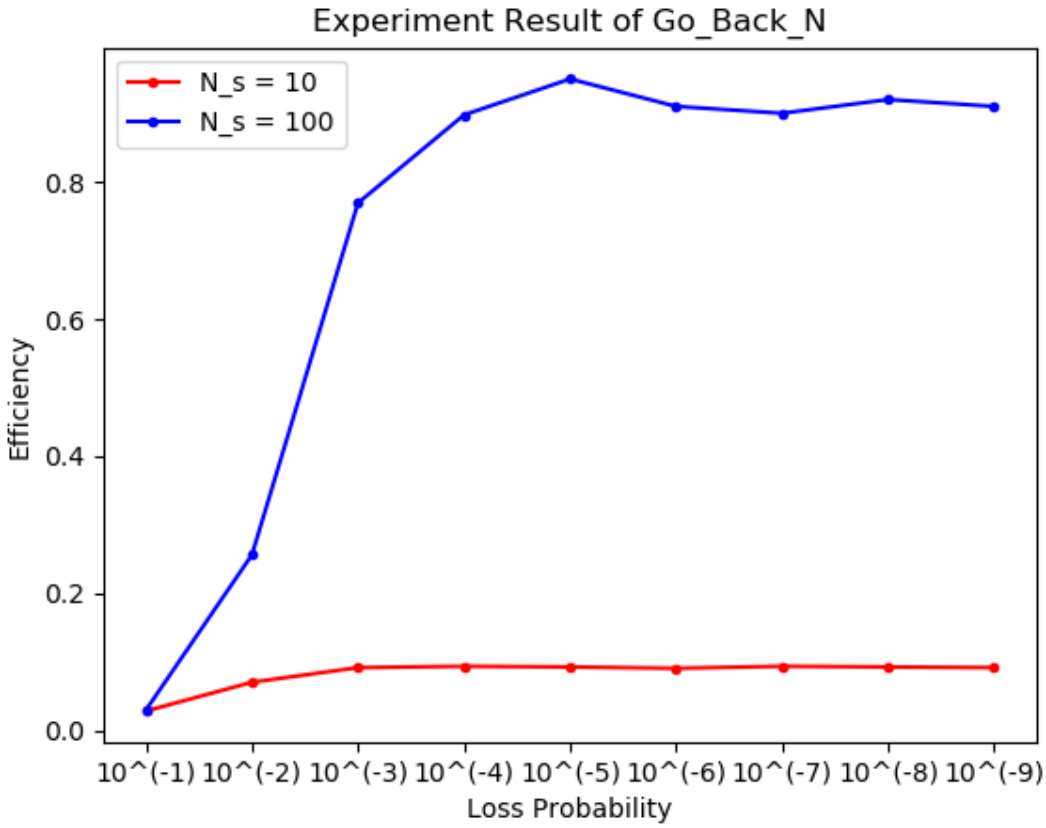
- 역시나 τ_1의 효율이 더 높았다.
- τ_1의 timeout 구간은 [11ms, 12ms]인 반면, τ_2의 timeout 구간은 [110ms, 120ms]이기 때문에, timeout 빈도 자체가 τ_2가 더 높고, t_m이 10배 차이 나기 때문에 그래프가 위와 같은 양상을 보인다고 추측할 수 있다.

WindowSize = 100 이고 RTT가 다를 때



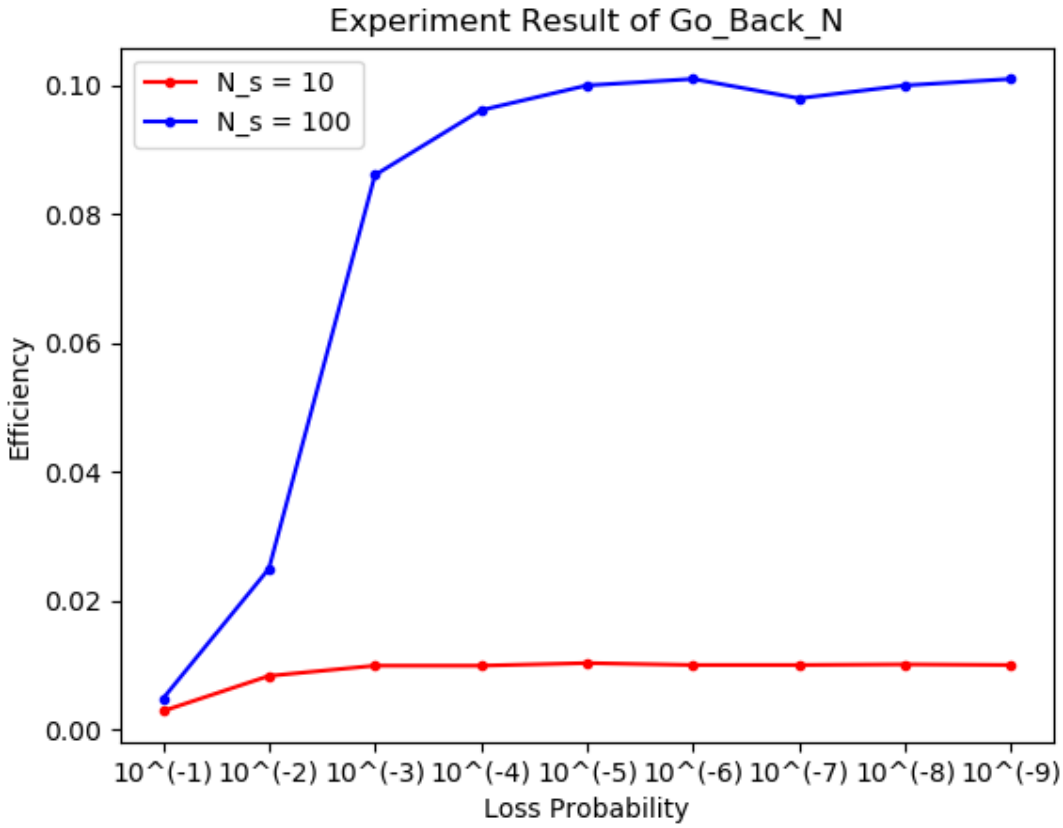
- go-back-N 전체에서, WindowSize = 100, $RTT(\tau) = \tau_1$일 때 가장 높은 효율을 보여주었다.

RTT = [8ms, 12ms] 이고 WindowSize가 다를 때



- 위 실험은 WindowSize가 클수록 효율이 높아진다는 것을 보여주는 단적인 예이다.
- 물론 WindowSize를 계속 늘리다 보면, 언젠가는 효율이 역행하는 모습이 나올 것이다. go-back-N의 특성상, 오류가 발생하면 Window 내의 패킷을 모두 재전송해야 하기 때문이다.

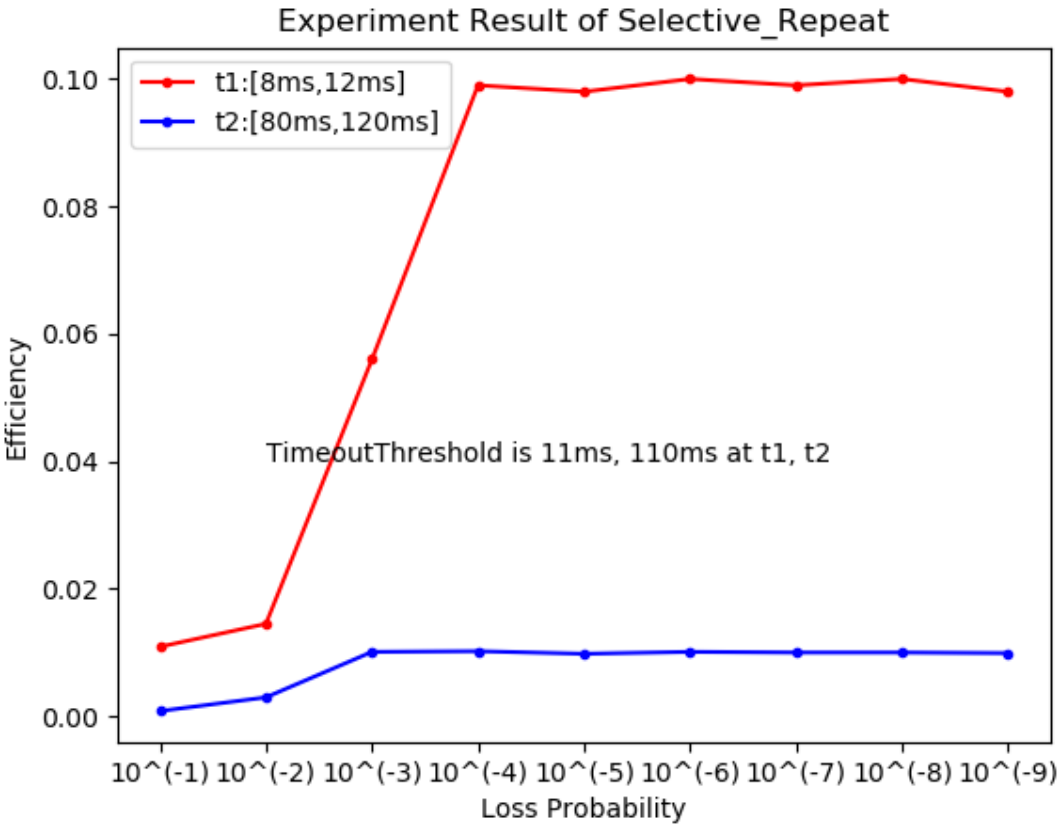
RTT = [80ms, 120ms] 이고 WindowSize가 다를 때



- 위 실험은 WindowSize가 클수록 효율이 높아진다는 것을 보여주는 단적인 예이다.
- 손실확률이 0으로 수렴하는 양상이라, 그래프도 일정 효율 값에 수렴하는 것처럼 보인다.

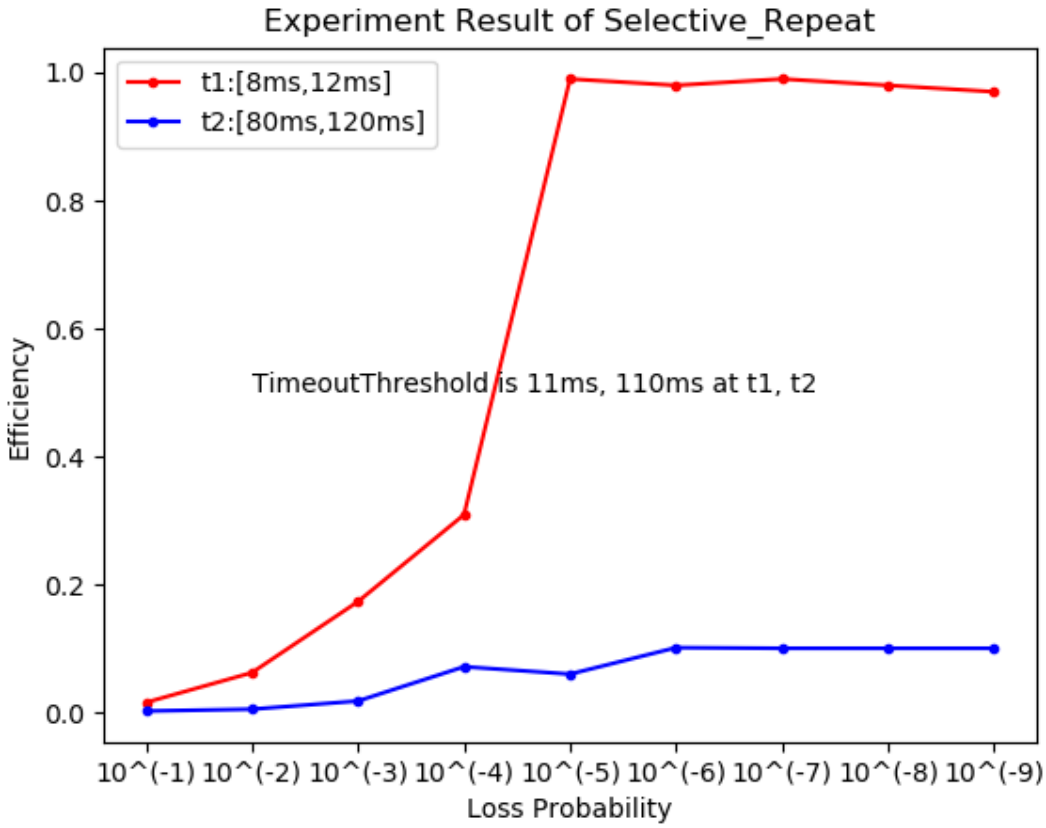
selective-repeat

WindowSize = 10 이고 RTT가 다를 때



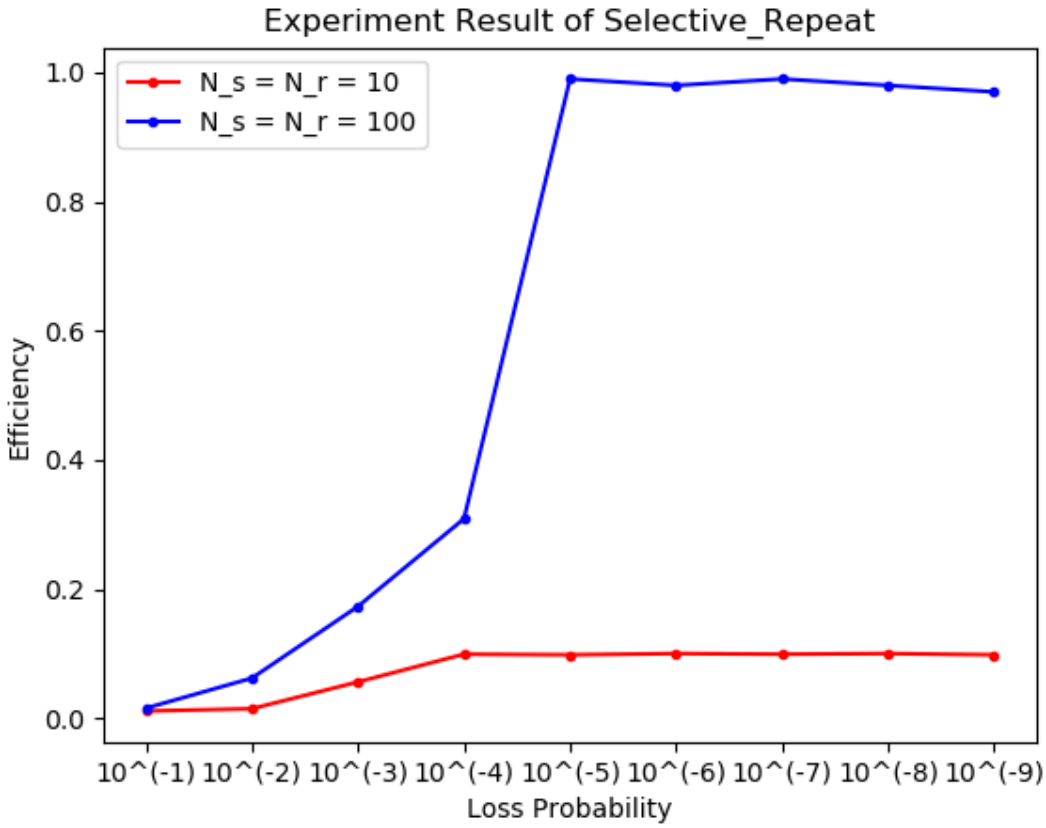
- go-back-N의 <WindowSize = 10이고 RTT가 다를 때> 와 비슷하다.
- go-back-N보다 효율성이 살짝 증가한 모습을 볼 수 있다.

WindowSize = 100 이고 RTT가 다를 때



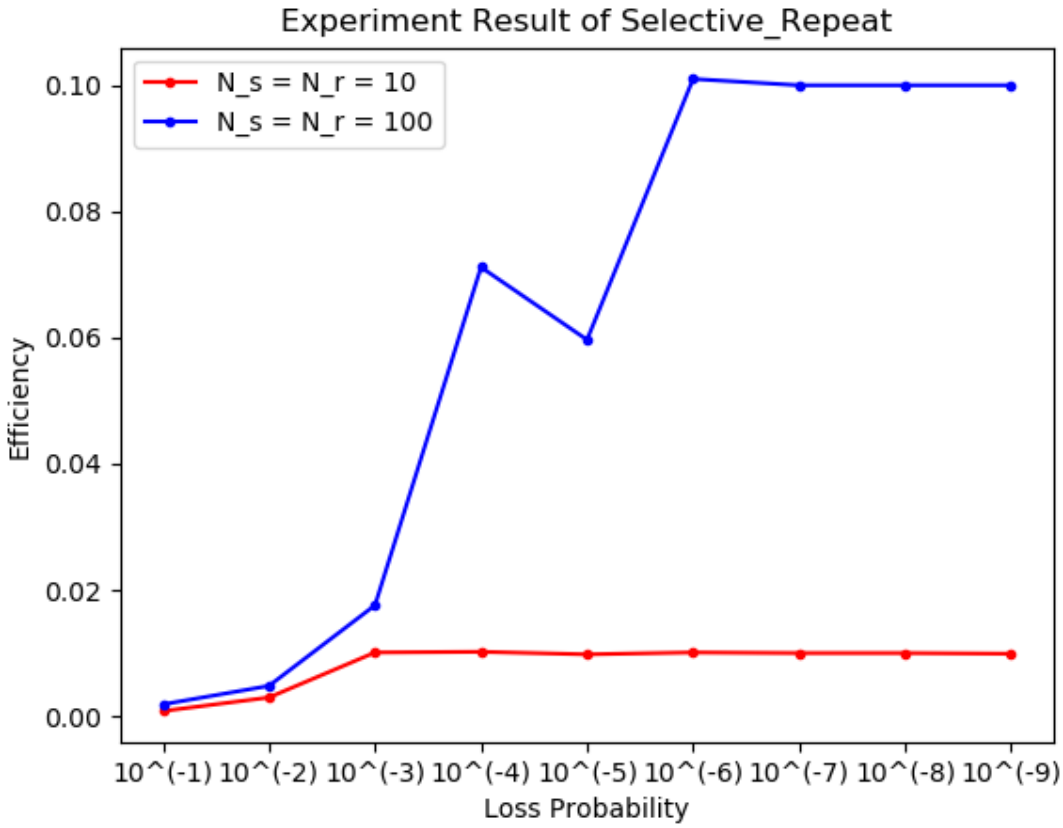
- selective-repeat 전체에서, WindowSize = 100, $RTT(\tau) = \tau_1$일 때 가장 높은 효율을 보여주었다. 효율성이 무려 1에 가까운 수치이다.

RTT = [8ms, 12ms] 이고 WindowSize가 다를 때



- 이 역시 WindowSize가 미치는 영향을 보여주는 단적인 예이다.
- selective-repeat은 실패한 패킷만 재전송하기 때문에, go-back-N보다는 효율이 역행하는 WindowSize의 임계점이 더 높을 것이다.

RTT = [80ms, 120ms] 이고 WindowSize가 다를 때



- 이 역시 WindowSize가 미치는 영향을 보여주는 단적인 예이다.
- 중간에 그래프가 크게 꺾였는데, timeout이나 loss가 많이 일어난 것 같다. 한 변인에 대하여 여러 번 측정 후 평균을 내어 그래프를 그렸으면 이런 현상이 줄어들 텐데, 시간상의 이유로 그러지 못한 점이 아쉽다.

Conclusion

- 다른 요인을 배제하고, 효율로만 따지면 rdt 3.0 <<< go-back-N < selective-repeat 순서의 퍼포먼스를 보여주었다.
- rdt 3.0의 경우, $RTT(\tau)$와 손실 확률이 프로토콜의 성능을 크게 좌우한다.
- go-back-N의 경우, 손실 확률이 낮을수록, WindowSize가 클수록 효율적인 측면이 강화된다. 그러나 WindowSize가 너무 커도 문제다. 효율성이 역행하는 현상이 생길 수 있으니, 적절한 WindowSize를 찾아 타협해야 한다.
- selective-repeat도 마찬가지로, 손실 확률이 낮을수록, WindowSize가 클수록 효율적인 측면이 강화된다. 이 역시 WindowSize가 매우 커졌을 때 문제가 생길 수 있

지만, 선택적인 재전송으로 go-back-N에 비해 불안정성이 크게 줄었다.

- go-back-N과 selective-repeat의 경우, $RTT(\tau)$값이 너무 작으면 *premature timeout*이 발생할 확률이 높아지고, 너무 크면 효율성이 급격히 떨어진다. 적절한 $RTT(\tau)$를 설정하였을 때, 최고의 효율을 뽑아낼 수 있다.

Reference

- Python Official documentation : _thread ⇒ Link
- Python Official documentation : Lock Objects ⇒ Link