- 과제 목표 (도출해야 할 결과)
  ➔ 단일프로세스 / 멀티프로세싱 / 멀티스레딩 통신부하 비교

- 코드 설명과 과제 해결 방법

  - 멀티프로세싱 / 멀티스레딩 대상 함수 : 같은 동작을 수행하고, 이름만 다르다. 대상 클라이언트 소켓에 대하여 수신하고, 답장을 전송한 후 소켓을 닫는다.

```
def multiprocess_sr(csocket, addr):

    # 1. Client로부터 데이터를 받음
    data = csocket.recv(1024)
    print(f"[Client {addr} Info] {data.decode()}")

    # 2. HTTP 200 OK, Content-type 전송
    res = "HTTP/1.1 200 OK\nContent-Type: text/html\n\n"
    csocket.send(res.encode('utf-8'))
    csocket.send(data)

    # 3. close
    csocket.close()
```

  - 멀티프로세싱 수행 내용 : Accept 수행 후 받은 클라이언트들의 소켓을 리스트로 모은 뒤, 순회하며 각 소켓마다 프로세스를 생성한다. Exception이 발생한 경우, 각 프로세스들의 종료를 join()을 통해 기다린다.

```
def main(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('', port))
    server_socket.listen()

    req_clients = list()
    try:
        # 반복 : accept -> recv -> send -> close
        while True:
            # 1. accept : 개별 클라이언트의 정보 받아옴 / csocket : 하나의 클라이언트
            (csocket, addr) = server_socket.accept()

            # 2. client list에 client 넣기
            req_clients.append(csocket)

            # 3. 요청 수 만큼 프로세스 생성(멀티프로세싱), 실행
            for req_client in req_clients:
                proc = Process(target=multiprocess_sr, args=(req_client, addr))
                proc.start()

            # 이미 배정된 클라이언트 리스트 초기화
            req_clients = list()

    except:
        proc.join()
```

- 멀티스레딩 수행 내용 : 클라이언트 소켓의 개별 처리에 프로세스 대신 스레드를 사용한다는 점 말고는, 전부 동일하다.

```
# 3. 요청 수 만큼 스레드 생성(멀티스레딩), 실행
for req_client in req_clients:
    th = Thread(target=multithread_sr, args=(req_client, addr))
    th.start()
```

- locustfile : 단순히 서버에 GET요청을 하기 위한 파일이며, locust의 수행 내용이다.
  wait_time이 1~5초 사이를 기다리도록 오버라이딩되고,
  task 데코레이터를 통해 task의 내용이 직접 정의한 index함수와 같이 수행된다.
  따라서 1~5초 사이의 간격을 두고 root 디렉토리에 대해 get을 수행하게 된다.

```
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    wait_time = between(1, 5)

    @task
    def index(self):
        self.client.get("/")
```

- 실행 결과 (공통적)

1. 성공 : 메시지를 잘 주고받음

```
[Client ('127.0.0.1', 62005) Info] GET / HTTP/1.1
Host: localhost:8890
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive


[Client ('127.0.0.1', 62011) Info] GET / HTTP/1.1
Host: localhost:8890
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive


[Client ('127.0.0.1', 63027) Info] GET / HTTP/1.1
Host: localhost:8890
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
```

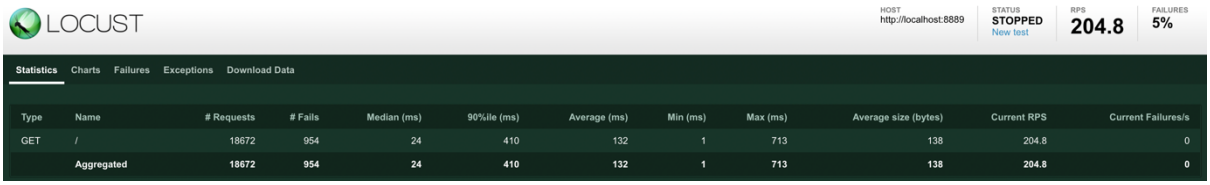
2. 실패 : Broken pipe (요청이 많을 경우 발생한 것으로 추정, multiprocessing에서 특히 많이 나타남)

```
  File "/Users/kamo/Documents/pythonProject/201702081_multiprocess.py", line 14,
 in multiprocess_sr
    csocket.send(data)
BrokenPipeError: [Errno 32] Broken pipe
```

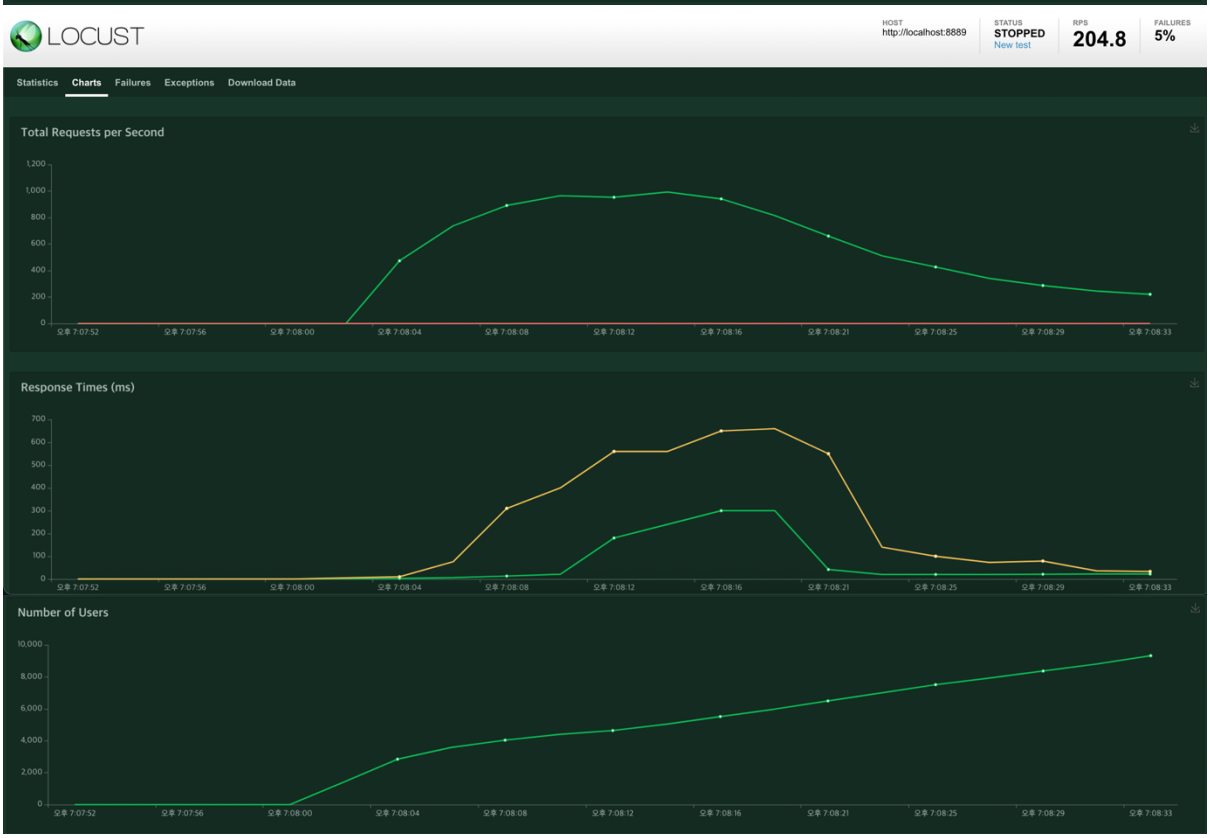
- Single Process vs Multi Process vs Multi Thread

  ➔ Total users : 10000, Hatch rate : 1000

  - Single Process : RPS가 204.8이며, 18672번의 요청을 했을 시 failure가 발생한다. 그리고 평균 응답시간은 132ms이다.

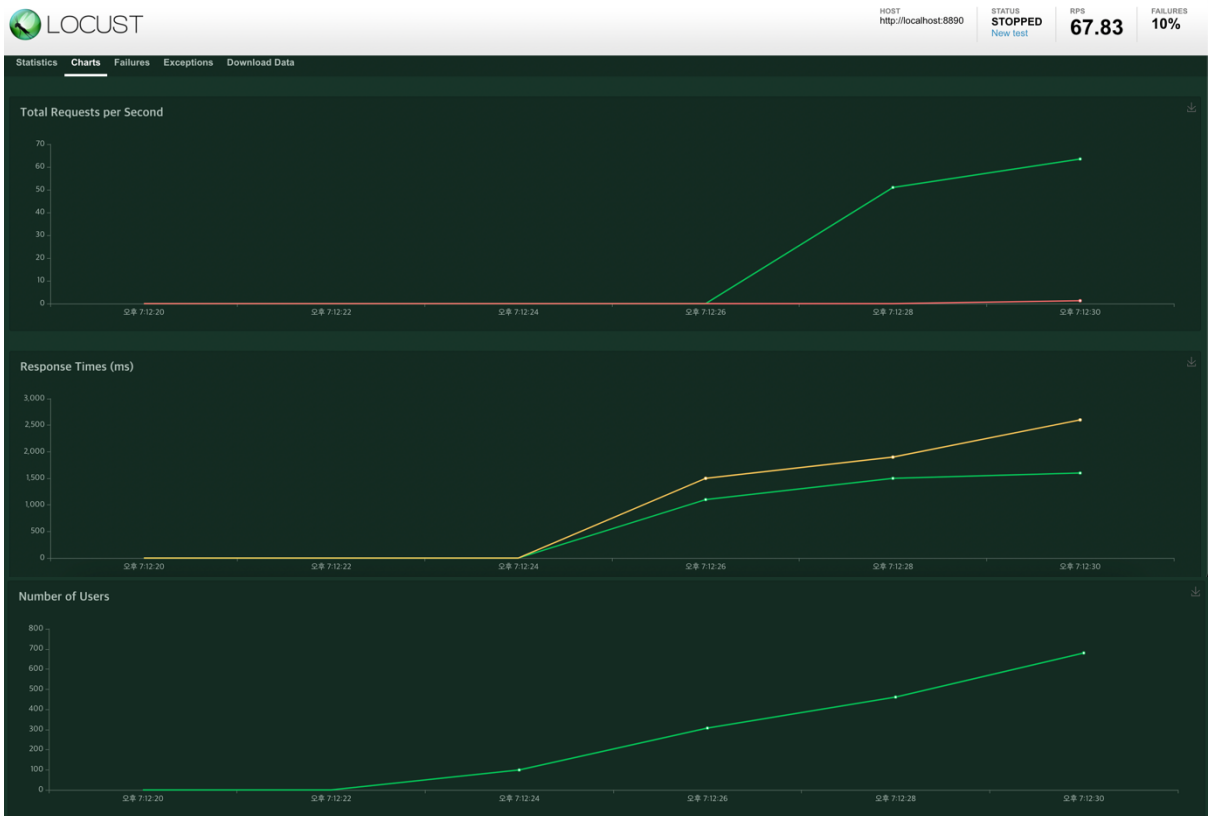


Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	18672	954	24	410	132	1	713	138	204.8	0
	Aggregated	18672	954	24	410	132	1	713	138	204.8	0



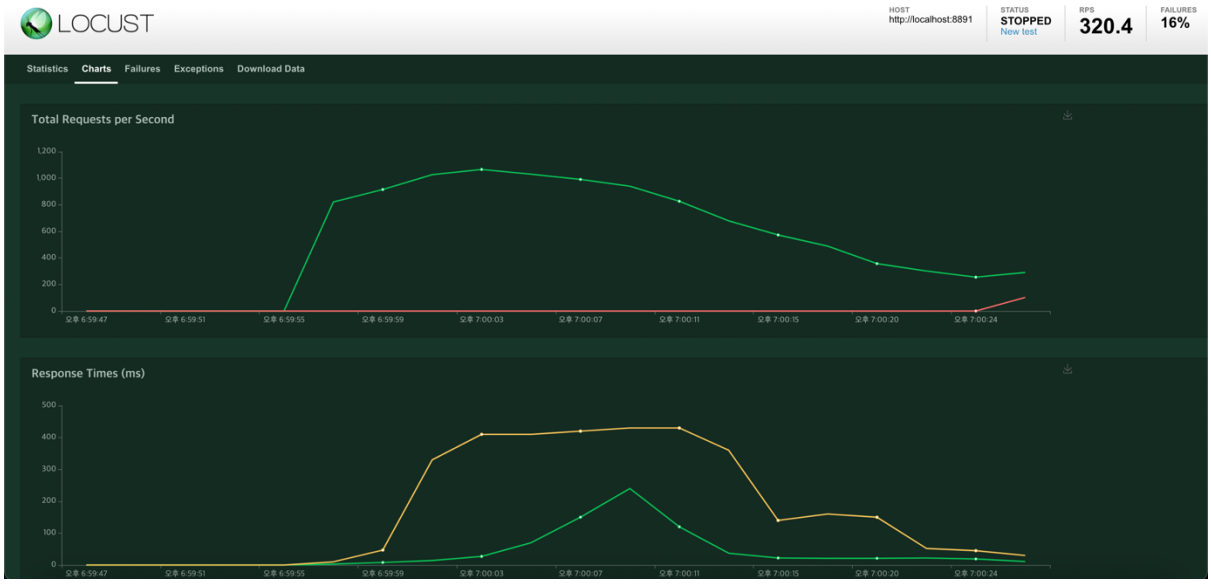
  - Multi Process : RPS가 67.83이며, 619번의 요청을 했을 시 failure가 발생한다. 그리고 평균 응답시간은 1656ms이다.

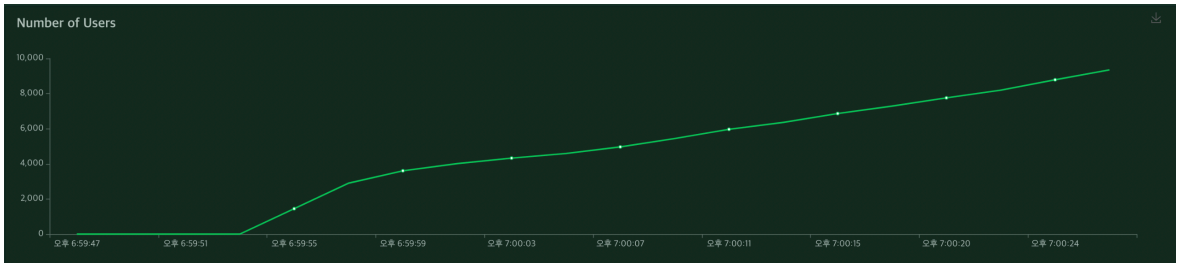
Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	619	64	1800	2900	1656	8	4161	130	67.83	3.83
	Aggregated	619	64	1800	2900	1656	8	4161	130	67.83	3.83



- Multi Thread : RPS가 320.4이며, 20522번의 요청을 했을 시 failure가 발생한다. 그리고 평균 응답시간은 88ms이다.

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	20522	3302	20	330	88	1	472	122	320.4	160.9
	Aggregated	**20522**	**3302**	**20**	**330**	**88**	**1**	**472**	**122**	**320.4**	**160.9**





- Multi Thread를 쓰는 방법이 가장 효율적이다. 이는 Request per Second가 가장 높고 응답시간도 가장 빠르며, Failure 까지 걸리는 Request 수도 가장 적어 안정적이기 때문이다.
- 반면에 Multi Process를 쓰는 방법은 Failure가 많이 발생하여 매우 불안정하였기 때문에, 이는 적절하지 않다는 생각이 들었다.

● 과제 느낀 점

컴퓨터가 자꾸 테스트 중에 리셋돼서 힘들었다..