- 과제 목표 (도출해야 할 결과)
  - ➔ Select를 이용한 서버의 구현
  - ➔ 단일프로세스 / 멀티프로세싱 / 멀티스레딩 / Select 통신부하 비교

- 코드 설명과 과제 해결 방법

  - 스레드 대상 함수 : 대상 클라이언트 소켓에 대하여 수신하고, 답장을 전송한 후 소켓을 닫는다. 수행 동작은 지난 주와 같다.

```
def multithread_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()
```

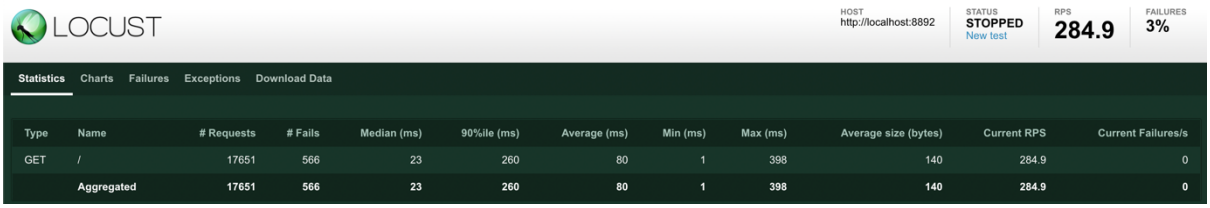
  - Main 수행 내용 : Accept 수행 후 받은 클라이언트들의 소켓을 리스트로 모은 뒤, Select를 사용하여 준비된 소켓들만 모은다. 이 후 준비된 소켓들을 순회하며 각 소켓마다 스레드를 생성한다. 스레드가 생성된 소켓들을 리스트에서 지운다. Exception이 발생한 경우, 각 스레드들의 종료를 join()을 통해 기다린다.

```
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 리스트를 Select 하여 처리, 실행
            input_ready, write_ready, except_ready = select.select(req_clients, [], [])
            for ready_client in input_ready:
                th = Thread(target=multithread_sr, args=(ready_client, addr))
                th.start()

            # Select하여 처리된 클라이언트 제외
            for done_client in input_ready:
                req_clients.remove(done_client)

    except:
        th.join()
```

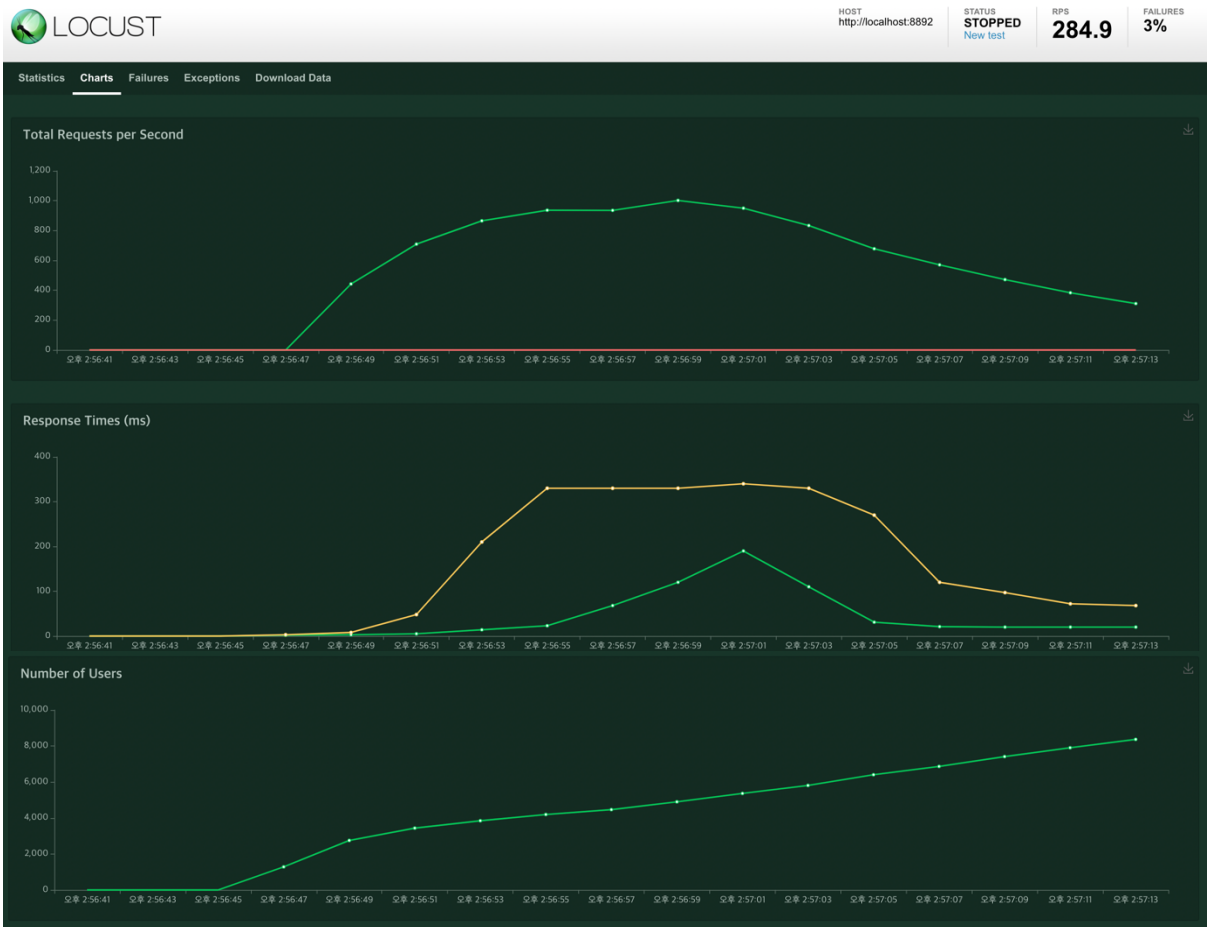
- locustfile은 지난 주와 같은 파일을 사용하였다.

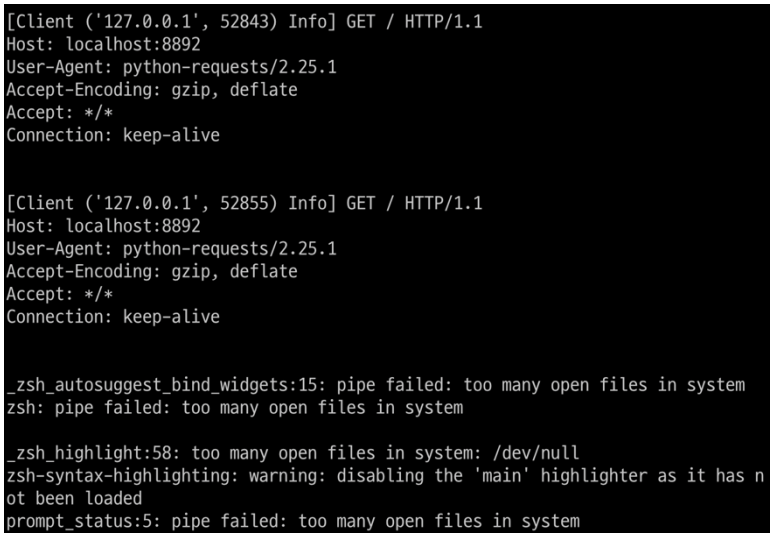
```
from locust import HttpUser, task, between

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

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

- 실행 결과



Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	17651	566	23	260	80	1	398	140	284.9	0
	Aggregated	**17651**	**566**	**23**	**260**	**80**	**1**	**398**	**140**	**284.9**	**0**





- Single Process vs Multi Process vs Multi Thread vs Select

Hatch rate : 1000, Total 10000 Requests	Request	Request per second	Average response time (ms)
Single process	18672	204.8	132
Multi process	619	67.83	1656
Multi thread	20522	320.4	88
Select	17651	284.9	80

- Request Per Second : Multi thread > Select > Single Process > Multi Process
- Average Response Time : Select < Multi thread < Single Process < Multi Process

➔ RPS는 Multi Thread가 가장 높았고, Select가 그 다음으로 높았다.
평균응답시간은 Select가 가장 짧았고, Multithread가 그 다음으로 짧았다.
Multi Process의 경우에는 전체적인 측면에서 매우 낮은 성능을 보였다.

RPS는 모든 방법에서 일정 클라이언트 수를 넘어서면 급격하게 떨어지는 양상을 보였기 때문에 Stop 버튼을 조금만 늦게 눌러도 차이가 벌어졌다. 따라서, 평균적인 응답시간이 더 중요한 척도라고 생각한다.
따라서 클라이언트에게 반응해주는 측면에서는 Select 방식이 가장 우수하다고 할 수 있다.

- 과제 느낀 점

컴퓨터가 자꾸 테스트 중에 리셋돼서 힘들었다..