Assignment 2: Web Server Imitation

2016314364 박수헌

개발 환경 : Windows 10 Home

Python 버전 : Python 3.8.8

사용한 IDE : conda를 사용한 spyder

사용한 라이브러리 : socket, os, threading

사용한 기기 : 랜선으로 연결돼 있는 데스크탑에서 server 코드를 실행한 후, 본 기기, 와이파이로 연결돼 있는 노트북, 와이파이에 연결돼 있는 휴대폰으로 접속 확인

Data structures and algorithm

```
serverPort = 10080
server = socket(AF_INET, SOCK_STREAM)
host = gethostbyname(getfqdn())
server.bind((host, serverPort))
print("website : http://"+ host + ":" + str(serverPort))
print('The TCP server is ready to receive.', host)
server.listen(5)
```

포트는 과제에서 지정한 대로 10080으로 설정했다.

AF_INET과 SOCK_STREAM을 사용해 server socket을 만들었다.

Host ip 주소를 알아내기 위해 우선 getfqdn()을 이용해 hostname을 알아낸 후, 이 값을 gethostbyname()에 전달해 ip 주소를 알아내었다. 현재 본인의 환경에서의 ip 주소는 192.168.35.223 이다.

이 host의 ip 주소와 port를 bind했다. 그리고 어떤 주소로 접속하면 되는지 print했다.

두 print 줄의 결과는 아래와 같았다.

```
website : http://192.168.35.223:10080
The TCP server is ready to receive. 192.168.35.223
```

server.listen(5)는 backlog를 5개까지 허용한다는 의미이다. 즉, 만든 서버 소켓으로 5개까지의 연결이 queue되게 허용한다는 의미이다. 최댓값은 보통 시스템에 다르지만 5인 경우가 많다. 그래서 5로 설정했다.

```
while True:
    client, (clientHost, clientPort) = server.accept()
    th = Thread(target = SendFile, args=(client,))
    th.start()
```

While True 루프를 이용해 무한히 실행되도록 설정했다.

우선 server가 연결을 요청하는 client를 허용한다. 그리고 그 리턴 값으로 client 소켓과, client의 ip 주소인 clientHost, client가 연결한 포트인 clientPort를 지정했다.

이런 client의 연결 요청이 다수에게서 올 수 있다. 이를 병렬적으로 처리하기 위해, 각 연결 당 하나의 thread를 만들어 실행한 후, start() 메소드를 통해 실행했다. 이런 방식으로 모든 client의 http 요청이 들어오는 대로 병렬적으로 실행하게 코딩했다.

Thread의 핵심인 SendFile이라는 함수는 인자로 client 소켓을 받는다. 이 함수는 아래와 같다.

```
def SendFile(client):
    files = os.listdir()
    received = client.recv(1024).decode()
    print('received : \n', received)
    received_split = received.split(' ')
    name, ext = os.path.splitext(received_split[1])
    name = name[1:]
```

우선 files라는 변수에 현재 디렉토리의 파일명들을 가진 list를 저장한다. 이는 다음과 같다.

```
['2016314364.docx',
 '2016314364.py',
 'jjambbong.png',
 'me.html',
 'ricenoodle.jpeg',
 'samgyup.jpeg',
 'steak.gif',
 'sushi.jpg']
```

received 라는 변수에는 서버가 클라이언트로부터 받은 내용의 1024 byte를 decode해서 담는다. 이 내용은 아래와 같다.

```
 GET /me.html HTTP/1.1
Host: 192.168.35.223:10080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: ko
```

GET 메서드를 이용해 서버에 요청을 보낸 것을 알 수 있다. 이 헤더를 살펴보면 많은 데이터를 담고 있다. 요청한 파일이 me.html이고, HTTP의 버전은 1.1이다. 그리고 Connection 부분을 보면 keep-alive라고 나와있다. HTTP 1.0 초기에는 요청이 올 때마다 TCP 연결을 맺어야 했다. 즉 예를 들어, 현재 본인의 html 파일에는 네 장의 이미지 파일이 포함되어 있다. 그러므로 persistent mode가 아니라면 html 파일과 네 장의 이미지 파일을 보내기 위해서 총 5번 클라이언트와 서버

를 통해 연결을 열고 닫는 과정이 필요하다는 의미이다. 이 때 Connection:keep-alive 헤더를 사용하면 한 번의 연결 안에서 여러 개의 객체를 보낼 수 있었다. 하지만 HTTP 1.1부터는 위의 헤더를 사용하지 않아도 기본적으로 persistent connection을 지원한다. 그래서 위에서 persistent mode에 대한 아무런 설정을 해주지 않고 그저 HTTP 1.1을 사용했을 뿐인데도 Connection:keep-alive라는 헤더가 포함되어 persistent mode로 작동하는 것을 볼 수 있다.

그 후 received_split에 받은 데이터를 공백을 기준으로 split했다. 그럼 나오는 list는 대략 ['GET', '/me.html', 'HTTP₩1.1' ...] 이런 형식의 list일 것이다. 이 때, 필요한 부분은 어떤 파일을 요청했는지 이므로 received_split[1]을 os.path.splitext를 이용해 파일 이름과 확장자로 나누어 줬다. 그럼 결과는 '/me' 와 '.html'일 것이다. 앞의 '/'는 필요 없으므로 다음 줄에서 name을 index 1부터만으로 다시 지정해주었다.

```
if name + ext in files:
    script = "HTTP/1.1 200 OK\r\n"
    if ext == '.html':
        filepath = name + ext
        contentType = 'text/html'
        with open(filepath, 'r') as f:
            script += "Content-Type: " + contentType + "\r\n"
            script += "\r\n"
            script += f.read()
            script += "\r\n\r\n"
            client.sendall(script.encode())
```

그 후, name + ext를 하면 위의 경우에서는 me.html일 것이고 다른 경우에는 sushi.jpg같은 형태가 될 것이다. 이 파일이 files라는 이전에 만든 디렉토리의 내 파일들의 list에 존재하는지 확인한다. 존재하는 경우에 위의 if문에 들어간다.

script라는 string에 client로 보낼 내용들을 담는다.

우선 연결이 정상적으로 돼야 하므로 200 OK 코드를 HTTP 버전과 함께 담고, ₩r₩n을 끝에 포함시킨다. 그 후, 두 가지의 경우로 나뉜다. 요청받은 파일이 html인지 이미지 파일인지에 따라 다르다. 위의 경우는 html일 경우이다. filepath에 name+ext, 즉 me.html을 담는다. 이는 html 파일이므로 content-type은 text/html로 지정해준다. With 구문을 이용해 객체를 open한 후, content-type을 지정하는 string과 ₩r₩n을 추가해 주고, html 파일의 내용을 f.read()를 통해 읽어 역시 script에 저장한다. 마지막으로 ₩r₩n을 추가한 후, 이 script를 encode해 client에게 보낸다. sendall의 인자는 byte만 담기 때문에 encode() 과정이 필요하다.

다음 코드이다.

```
else :
    filepath = name + ext
    contentType = 'image/' + ext[1:]
    with open(filepath, 'rb') as f:
        script += "Content-Type: " + contentType + "\r\n"
        script += "\r\n"
        client.sendall(script.encode())
        bdata = f.read()
        client.sendall(bdata)
        client.sendall("\r\n\r\n".encode())
```

위의 else문은 요청받은 파일의 확장자가 html이 아닐 경우, 즉 이미지 파일인 경우이다. 이때 위의 html 파일의 경우와 크게 다른 점은 content-type이다. Content-type을 파일에 따라 image/jpg, image/gif, image/jpeg 등등으로 지정해주어야 한다. 그러므로 확장자의 정보를 담고있는 ext를 이용해 content-type을 지정해주었다. 그 후, with 구문을 통해 binary 형식으로 이미지 파일을 읽었다. 그렇게 되면 client에게 sendall해줄 때에 encode를 해주지 않아도 된다. 어차피 binary 형식이기 때문이다. 그 이외의 과정은 html을 보낼 때와 같다.

```
else:
    client.sendall('HTTP/1.1 404 Not Found\n'.encode())
client.close()
```

이 else문은 위에 있던 if name+ext in files: 가 아닌 경우이다. 즉 디렉토리 내에 존재하지 않는 파일을 요청했을 때의 경우이다. 이때는 404 Not Found 에러를 송출해야한다. 따라서 위의 200 OK 코드가 아닌 404 Not Found를 사용했다. "favicon.ico"같은 파일을 요청했을 경우의 결과화면은 아래와 같다.

http://192.168.35.223:10080/favicon.ico 를 입력했다.

작동을 확인하기 위해 작성한 html 파일은 아래와 같다.

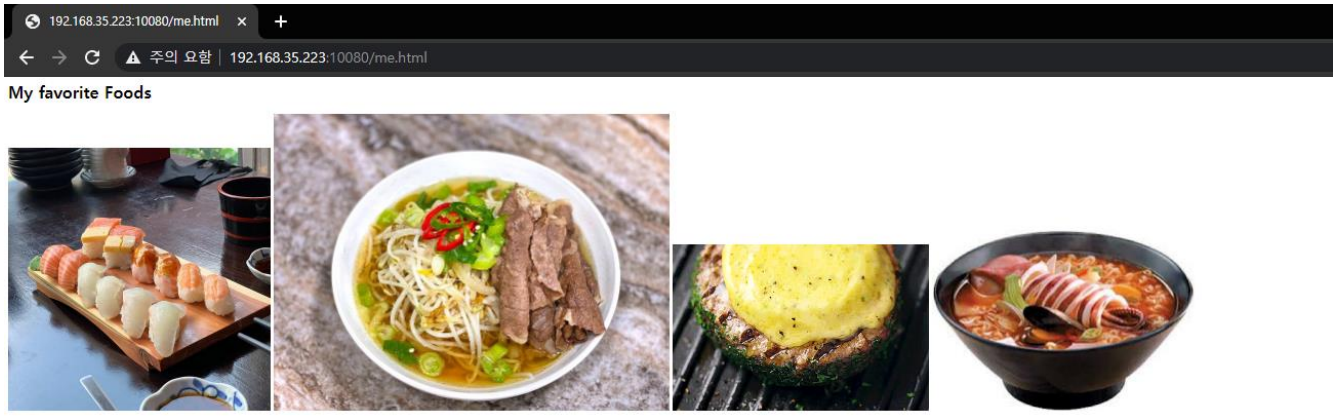
```
<html>
    <body>
        <h1>My favorite Foods</h1>
        <p>
            <img src="sushi.jpg"
                alt="sushi"/>
            <img src="ricenoodle.jpeg"
                alt="ricenoodle"/>
            <img src="steak.gif"
                alt="steak"/>
            <img src="jjambbong.png"
                alt="jjambbong"/>

        </p>
    </body>
</html>
```

우선 body에 "My favorite Foods"라는 텍스트를 넣었다.

다음, 새로운 paragraph에 총 4장의 사진을 넣었다. 각각 jpg, jpeg, gif, png이다. 모든 형식의 이미지 파일이 잘 보여지는 지 확인하기 위해서이다.

Results

http://192.168.35.223:10080/me.html 의 결과이다.

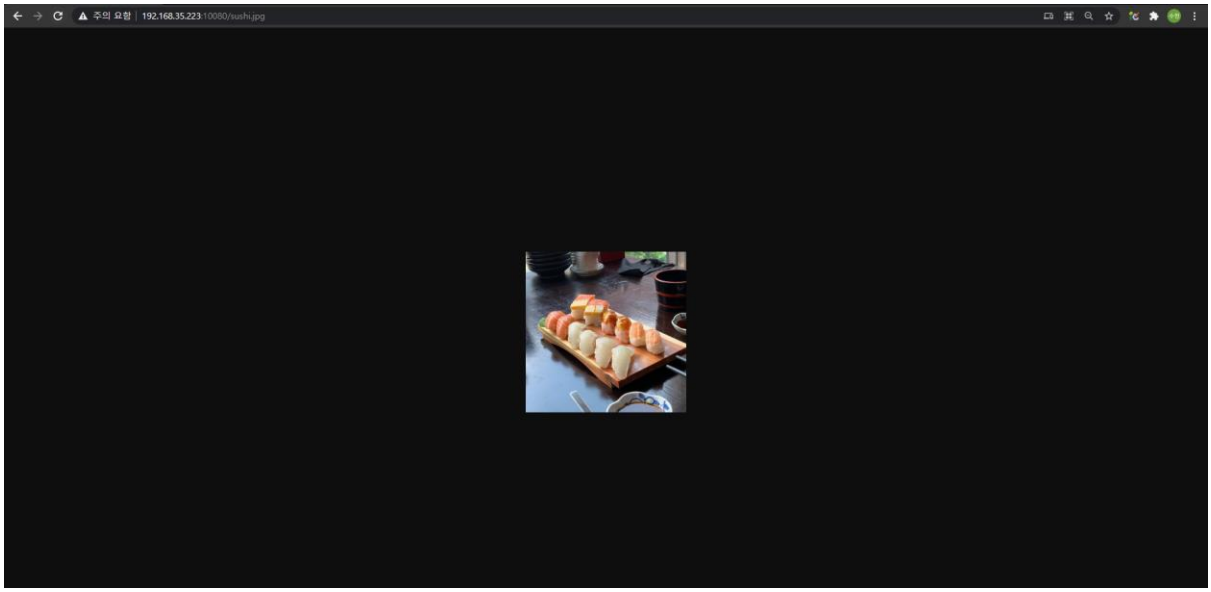


맨 위에 text와 네 장의 이미지가 문제없이 표현되는 것을 볼 수 있었다. Gif 파일도 문제없이 재생되었다.

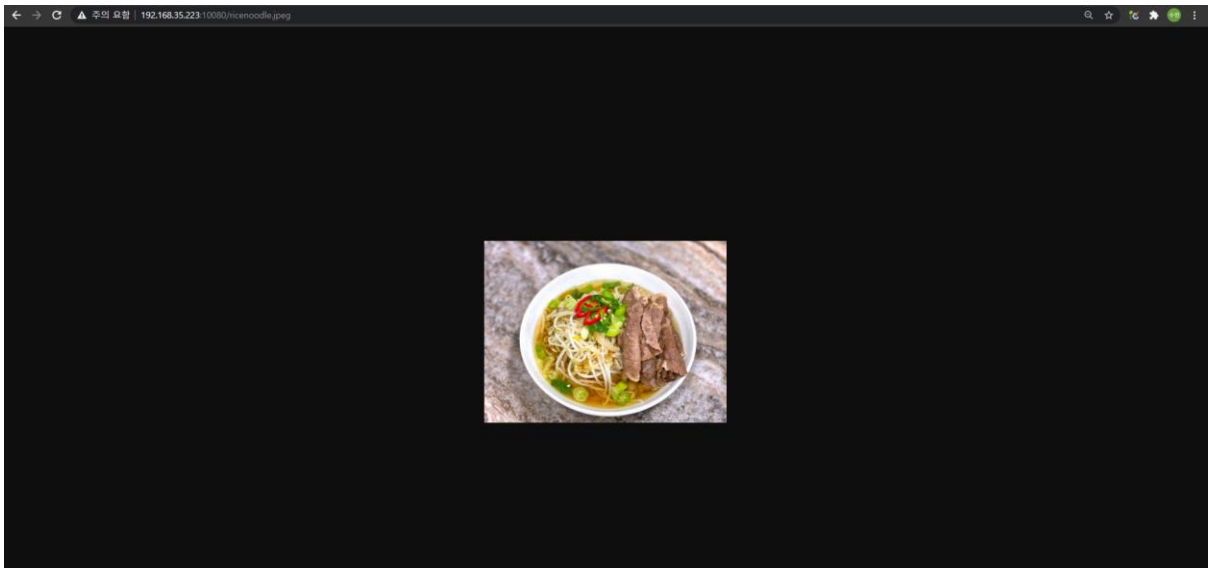
아래는 receive되는 get 메소드의 첫 부분만을 print하도록 코드를 수정한 후, 여러 머신에서 서버에 반복적으로 접속했을 때의 결과 화면이다.

아래는 이미지 파일만을 요청했을 때의 화면이다.

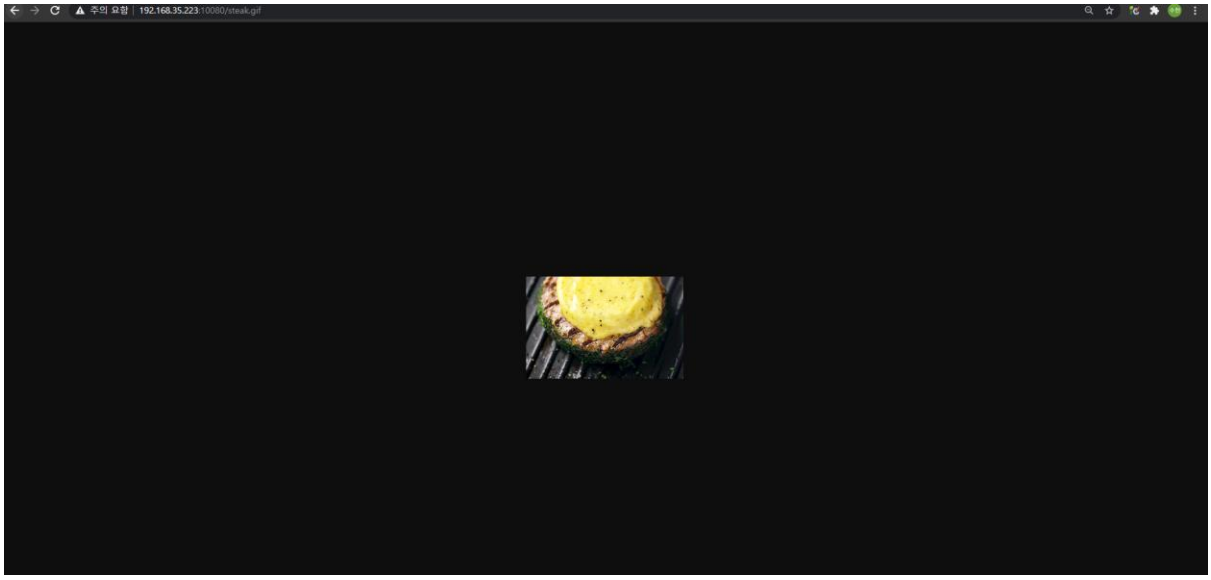
1. http://192.168.35.223:10080/sushi.jpg



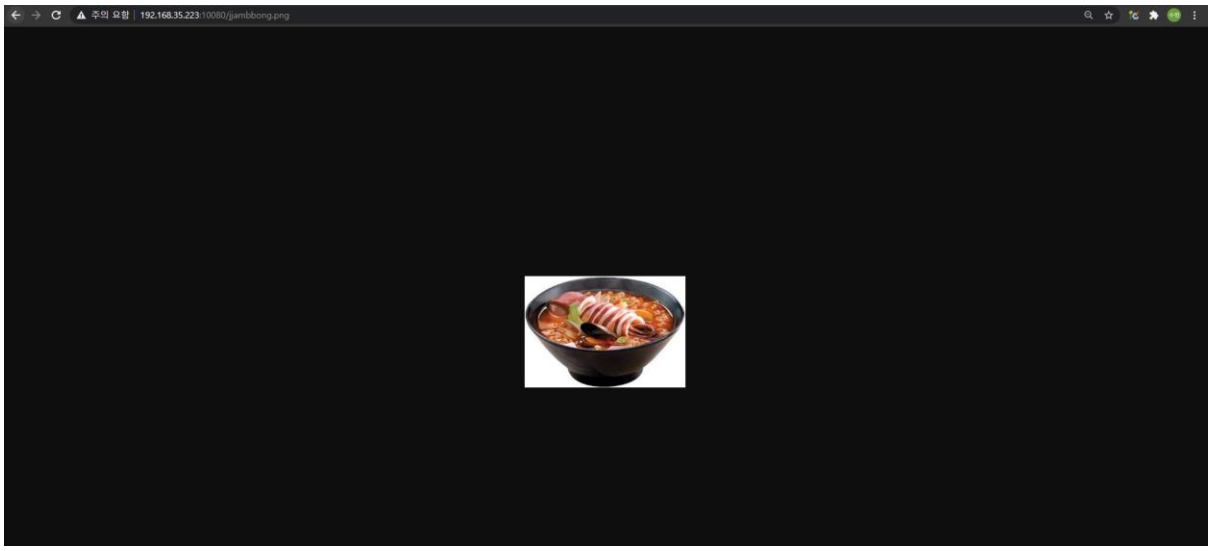
2. http://192.168.35.223:10080/ricenoodle.jpeg



3. http://192.168.35.223:10080/steak.gif

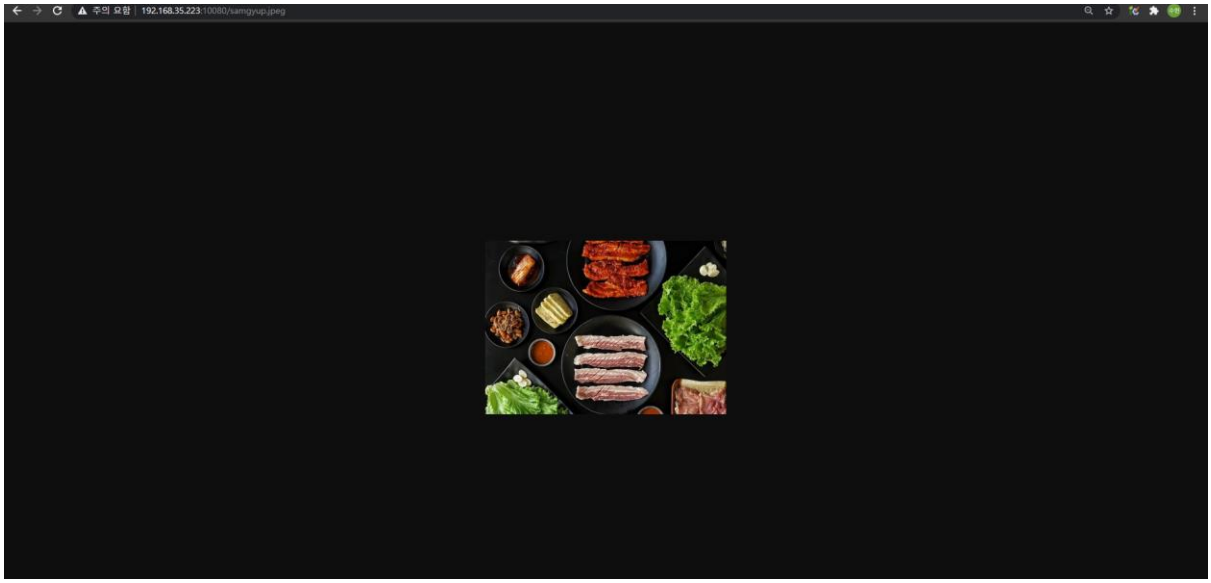


1. http://192.168.35.223:10080/jjambbong.png



다음은 me.html 파일에 포함되어 있진 않지만 디렉토리 내에 존재하는 samgyup.jpeg 파일에 대한 요청 결과이다.

http://192.168.35.223:10080/samgyup.jpeg



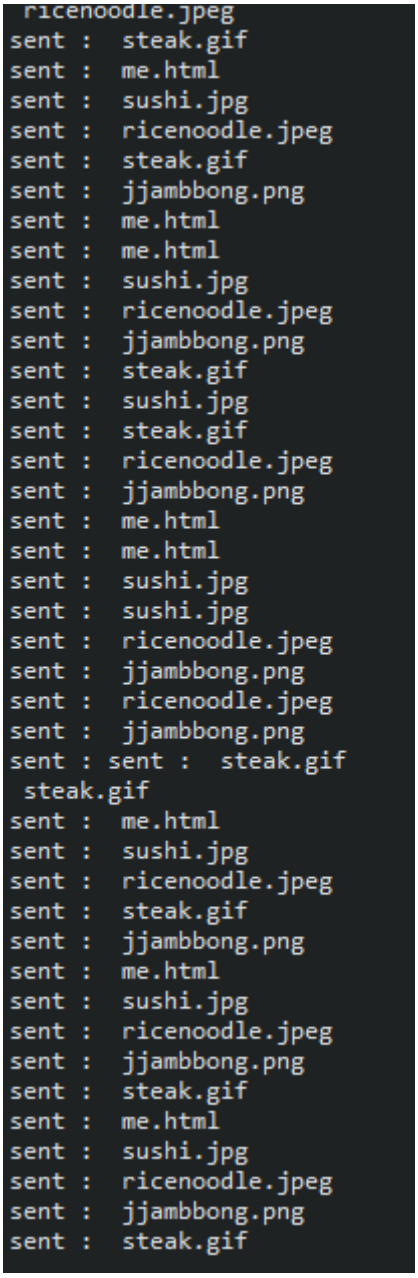
```
website : http://192.168.35.223:10080
The TCP server is ready to receive. 192.168.35.223
received :
 ['GET', '/me.html', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/sushi.jpg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/ricenoodle.jpeg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/steak.gif', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/jjambbong.png', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/me.html', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/sushi.jpg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/ricenoodle.jpeg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/steak.gif', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/jjambbong.png', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/me.html', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/sushi.jpg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/ricenoodle.jpeg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/steak.gif', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/jjambbong.png', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/me.html', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/sushi.jpg', 'HTTP/1.1\r\nHost:']
received :
 ['GET', '/ricenoodle.jpeg', 'HTTP/1.1\r\nHost:']
```

크롬에서 우선 html 파일을 받으면 자동으로 그 파일안에 포함된 이미지 파일들을 요청한다. 따라서 me.html이 요청된 후, 순서대로 그 안에 포함된 이미지 파일들을 요청하는 것을 볼 수 있다.

하지만 여러 기기에서 매우 짧은 간격으로 새로고침을 해보면 아래와 같은 콘솔 화면이 나온다.

Thread를 사용한 결과이다. 순서대로 me.html, sushi, ricenoodle, steak, jjambbong을 요청받고 보낸 후에 다시 이 순서대로 요청을 받는 것이 아니라, 병렬적으로 작동되므로 크기가 큰 파일들은 조금 느리게 보내지고 크기가 작은 파일은 빠르게 보내지므로 이렇게 순서가 달라지는 모습을 볼 수 있다.

파일이 완전히 보내지면 확인 문구만을 print하게 코드를 수정한 후 똑같이 실험해 보았다.



위의 get 메소드와 같이 순서가 꼬인 채로 보내지는 것을 볼 수 있다. 이 또한 http 요청을 병렬적으로 처리한 결과라고 볼 수 있다.