CS339 Lab3

邬心远

519021910604

Environment

- python == 3.8
- OS: Ubuntu20.04

C/S model

Implementation of client and server

The `socket` package is used to realize the function of client and server model, they are implemented seperately in `client.py` and `server.py`.

The major codes are as follows:

```
## server:
def server():
    server_port = 12000
    server_socket = socket(AF_INET, SOCK_STREAM)
    server_socket.bind(('', server_port))
    server_socket.listen(6)
    while True:
        sock, addr = server_socket.accept()
        t = threading.Thread(target=tcpLink, args=(sock, addr))
        t.start()

## client:
def client(client_name, server_ip='127.0.0.1'):
    server_port = 12000
    client_socket = socket(AF_INET, SOCK_STREAM)
    client_socket.connect((server_ip, server_port))
    total_len = int(client_socket.recv(1024).decode())
    recv_len = 0
    sentence = 'request for file'.encode()
    client_socket.send(sentence)
    file = open('recv/'+client_name+'.txt', 'w')
    while recv_len != total_len:
        recv = client_socket.recv(1024)
        recv_len += len(recv)
        file.write(recv.decode())
    recv_time = time.time()
    client_socket.send(str(recv_time).encode())
    client_socket.close()
```

Specially, `SOCK_STREAM` set the connection to be TCP, and client can connect to server through `connect(ip, port)`. The client then send a request to server, and then store the data it receive in file `/recv/h1.txt` if its name is `h1`.

For server, it waits for the connection from client and once hear something then create a thread to serve a client. In the thread, it send the file to client socket where the message is from.

Using mininet
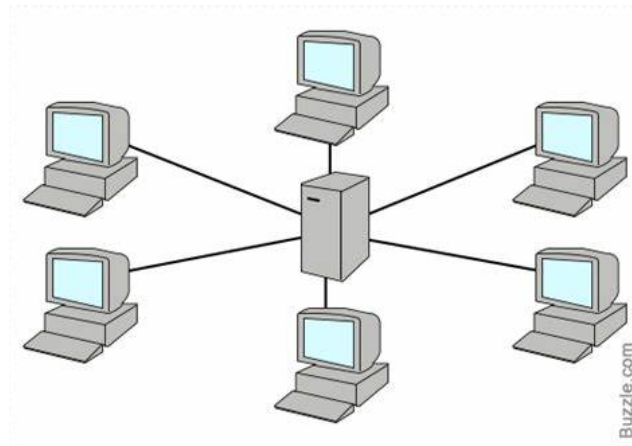
The above topology can be structured using mininet like the work done in previous lab. The code are in `CS.py`.

The host in the middle is treated as router.

Function in the former part are call like `r1.cmd('python3 server.py >  server.log &')`. This can also write the output into a log file.

Downloading time

To measure the total downloading time, we can use `time.time()` to get the time start tcp link and the time finish downloading. The former one is get by server and the latter one can be get by client and then sent to server. Here is the TCPlink function in server:

```python
def tcpLink(sock, addr):
    global start_time
    if start_time == 0:  # means it's the first time tcp
        start_time = time.time()
    print('Accept new connection from %s:%s...' % addr)
    file = open('data.txt', 'r')
    send = file.read().encode()
    send_len = len(send)
    sock.send(str(send_len).encode())
    data = sock.recv(1024).decode()
    if data == 'request for file':
        sock.send(send)
    data = sock.recv(1024).decode()  # time of receiving data
    print('{} seconds past before {} receives data'.format(float(data)-
start_time, addr))
    sock.close()
    print('Connection from %s:%s closed.' % addr)
```

and the out put is like:

```
Accept new connection from 192.168.10.1:42162...
0.0012135505676269531 seconds past before ('192.168.10.1', 42162) receives data
Connection from 192.168.10.1:42162 closed.
```

```
Accept new connection from 192.168.20.1:36682...
0.14070582389831543 seconds past before ('192.168.20.1', 36682) receives data
Connection from 192.168.20.1:36682 closed.
Accept new connection from 192.168.30.1:51724...
0.2676823139190674 seconds past before ('192.168.30.1', 51724) receives data
Connection from 192.168.30.1:51724 closed.
Accept new connection from 192.168.40.1:54208...
0.41979312896728516 seconds past before ('192.168.40.1', 54208) receives data
Connection from 192.168.40.1:54208 closed.
Accept new connection from 192.168.50.1:42104...
0.5772209167480469 seconds past before ('192.168.50.1', 42104) receives data
Connection from 192.168.50.1:42104 closed.
Accept new connection from 192.168.60.1:54588...
0.7377035617828369 seconds past before ('192.168.60.1', 54588) receives data
Connection from 192.168.60.1:54588 closed.
```

We can see that the time increase linearly as more request it receives.

Problems and solution

When I tried to send big data, using the following codes, I found sometimes the client can't receive complete file:

```
## server:
file = open('data.txt', 'r')
sock.send((file.read().encode()))

## client:
file = open('recv/'+client_name+'.txt', 'w')
while True:
    recv = client_socket.recv(1024)
    print(len(recv))
    file.write(recv.decode())
    if len(recv) < 1024:
        break
```

Name	Size	Modified	
h1.txt	155.7 kB	13:45	☆
h2.txt	155.7 kB	13:45	☆
h3.txt	90.0 kB	13:45	☆
h4.txt	155.7 kB	13:45	☆
h5.txt	155.7 kB	13:45	☆
h6.txt	155.7 kB	13:45	☆

I searched online, and guess it may caused by server sending slower than client, so I choose to send the length of file before sending the content, codes are modified to:

```
## server:
file = open('data.txt', 'r')
send = file.read().encode()
send_len = len(send)
```

```
sock.send(str(send_len).encode())
data = sock.recv(1024).decode()
if data == 'request for file':
sock.send(send)

## client:
total_len = int(client_socket.recv(1024).decode())
    recv_len = 0
    sentence = 'request for file'.encode()
    client_socket.send(sentence)
    file = open('recv/'+client_name+'.txt', 'w')
    while recv_len != total_len:
        recv = client_socket.recv(1024)
        recv_len += len(recv)
        file.write(recv.decode())
```

After this change, the file received is complete.

Name		Size	Modified	Star
h1.txt		155.7 kB	14:23	☆
h2.txt		155.7 kB	14:23	☆
h3.txt		155.7 kB	14:23	☆
h4.txt		155.7 kB	14:23	☆
h5.txt		155.7 kB	14:23	☆
h6.txt		155.7 kB	14:23	☆