Lab3: Socket Programming

** Xin Xu 519021910726 **

Environment

** g++ 9.3.0 **
** python 3.8 **
** Ubuntu 20.04 **

C/S Model

Implement C/S Model

According to the question, I use multithreads to simulate server handling several requests at the same time and several clients request a file transfer simultaneously.

For the server, there are fixed number of threads to deal with file transfer just like a threading pool.

```
pthread_t servers[NUM_OF_SERVER_THREAD];
    int index_s[NUM_OF_SERVER_THREAD];

    for(int i = 0; i < NUM_OF_SERVER_THREAD; ++i) {
        index_s[i] = i;
        pthread_create(&servers[i], NULL, &Server_thread, (void*)&(index_s[i]));
    }
```

The `index` array is to seperate the address of the identifier `i`.

Since there are multiple threads, we need a mutex lock:

```
pthread_mutex_lock(&mutex);
        answer_sock = accept(welcome_sock, (struct sockaddr*)&Client_ADD,
&Client_ADD_SIZE);
        printf("\nHi,I am running server thread %d.Some Client hit me\n", id);
        pthread_mutex_unlock(&mutex);
```

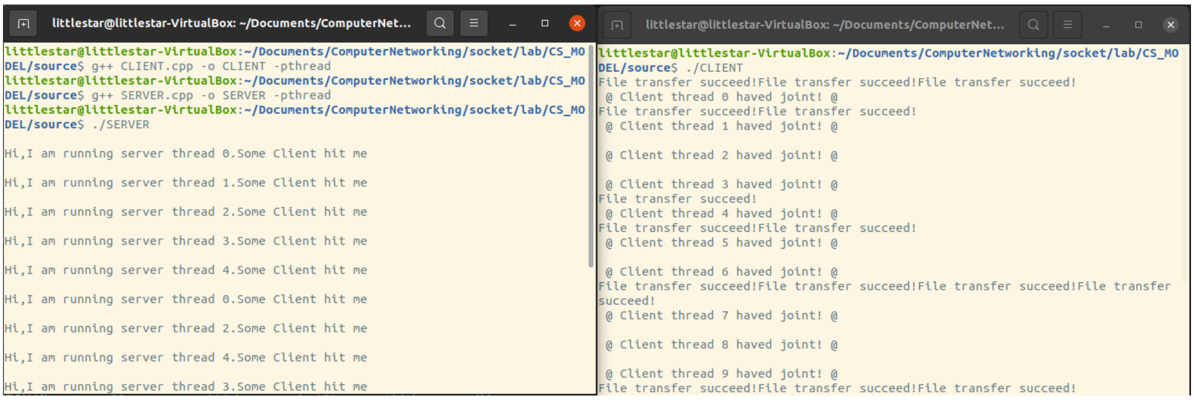
We don't the size of the file to be transferred, so we can only transfer the file bit by bit:

```
int n = 0;
        while((n = fread(datasending, 1, MAX_DATA, file_fp)) > 0) {
            if(write(answer_sock, datasending, n) < 0) {
                printf("\nThe connect is shutdown! Cannot write!");
                pthread_exit(0);
            }
        }
        shutdown(answer_sock, SHUT_WR);
        fclose(file_fp);
```

For clients, we also receive the file bit by bit in case of big file:

```
int n = 0;
    while((n = read(Client_sock, datareceived, sizeof(datareceived) - 1)) > 0) {
        fwrite(datareceived, n, 1, fp);
    }
    printf("File transfer succeed!");
```

To run this two file, you just need to open the terminal in the directory, compile them with commands `g++ CLIENT.cpp -o CLIENT -pthread` and `g++ SERVER.cpp -o SERVER -pthread`. Finally run `./SERVER` and `./CLIENT` in order. Process and results shows as follows:



After running, there are many `.txt` file in this directory.

Use Mininet to Compare the Overall File Downloading Time

At first, I don't know we can run `.cpp` file in mininet. So, I write a `.py` file again. In this time, in order to match mininet, the strategy of multithreads changes. For server, every time it `accept` a request, server will create a new thread to deal with it. For client, I quit multithreads for in reality, chances are that multiple hosts request a file once rather than one host requests a file multiple times.

The codes of multithreads in server:

```
answer_sock, client_addr = welcome_sock.accept()
        print('accept a client with ip address of {}!'.format(client_addr))

        thread_server = threading.Thread(target= file_transfer, args=
(answer_sock,client_addr))
        thread_server.setDaemon(True)
        thread_server.start()
```

We can see that there is no mutex lock, and we don't need to maintain the state of these threads. It is more easier.

The folder `client1` to `client6` is to simulate different catalog in different hosts.

To simulate in mininet, you can run `CS_topo.py` with command `sudo python3 CS_topo.py`. Processes and results are as follows:

```
littlestar@littlestar-VirtualBox:~/Documents/ComputerNetworking/socket/lab/CS_MO
DEL/source$ sudo python3 CS_topo.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server client1 client2 client3 client4 client5 client6
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** client1 : ('python3 client.py -n client1 -t 5 &',)
[1] 22578
*** client2 : ('python3 client.py -n client2 -t 5 &',)
*** client3 : ('python3 client.py -n client3 -t 5 &',)
*** client4 : ('python3 client.py -n client4 -t 5 &',)
*** client4 : ('python3 client.py -n client5 -t 5 &',)
*** client6 : ('python3 client.py -n client6 -t 5 &',)
*** client1 : ('python3 client.py -n client1 -t 4 &',)
[2] 22584
*** client2 : ('python3 client.py -n client2 -t 4 &',)
```

The time for each client to download is in the file `server.log`:

```
server ready

accept a client with ip address of ('10.0.0.4', 59540)!
File transfer successfully! It spend 0.0004999637603759766 seconds to transfer to
client ('10.0.0.4', 59540)

accept a client with ip address of ('10.0.0.2', 51034)!
File transfer successfully! It spend 0.0004029273986816406 seconds to transfer to
client ('10.0.0.2', 51034)

accept a client with ip address of ('10.0.0.2', 51036)!
File transfer successfully! It spend 0.0005345344543457031 seconds to transfer to
client ('10.0.0.2', 51036)

accept a client with ip address of ('10.0.0.3', 35634)!
File transfer successfully! It spend 0.0007405281066894531 seconds to transfer to
client ('10.0.0.3', 35634)

......
```

To observe the trend of download time with the number of hosts increasing, firstly we look at the `CS_topo.py` file:

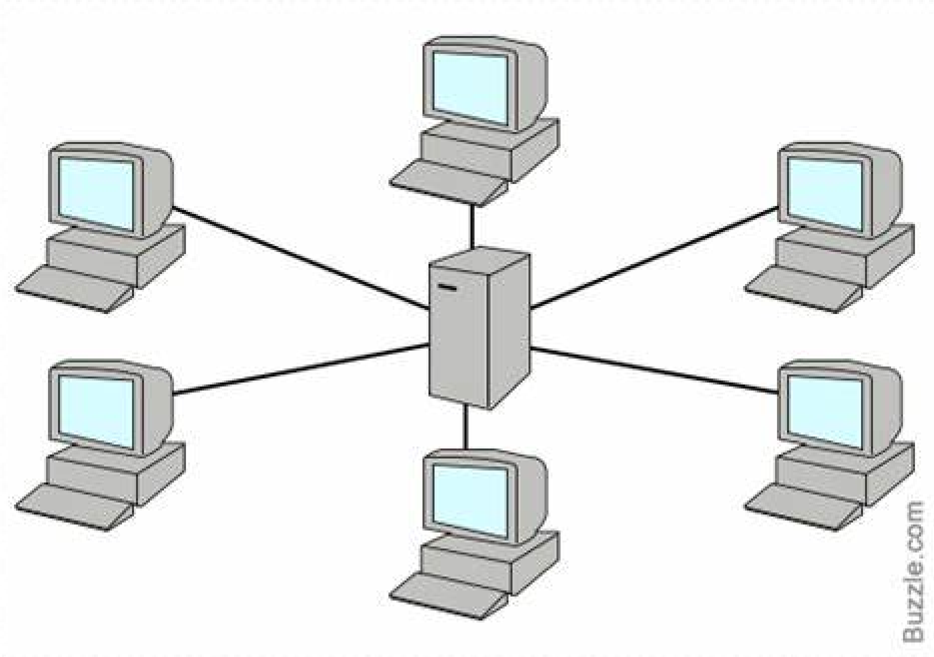
```
server.cmd('python3 -u server.py > server.log &')
    time.sleep(3)

    i = 5
    while i > 0:
        client1.cmdPrint('python3 client.py -n client1 -t %d &' % i)
        client2.cmdPrint('python3 client.py -n client2 -t %d &' % i)
        client3.cmdPrint('python3 client.py -n client3 -t %d &' % i)
        client4.cmdPrint('python3 client.py -n client4 -t %d &' % i)
        client4.cmdPrint('python3 client.py -n client5 -t %d &' % i)
        client6.cmdPrint('python3 client.py -n client6 -t %d &' % i)
        i = i - 1
```

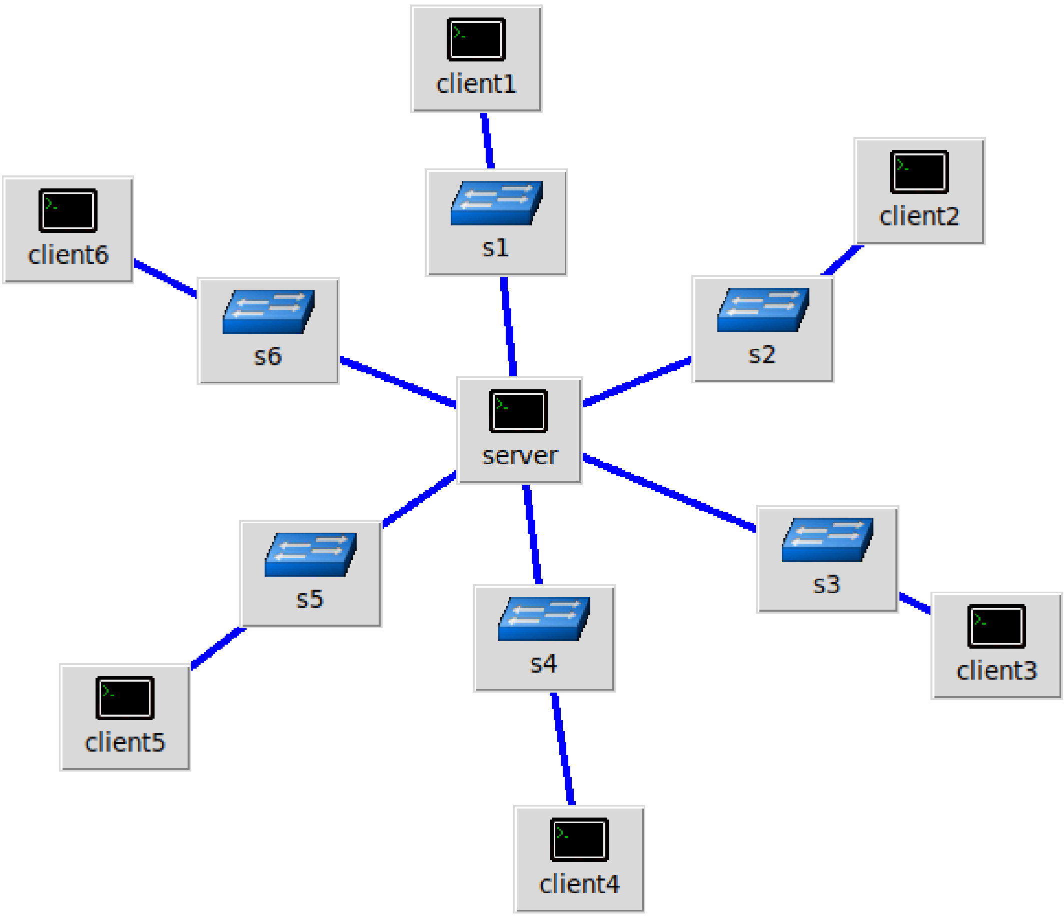
We can increase the value of `i` to simulate the increasing of hosts number. As we can see from `server.log`, with the number of hosts increases, the downloading time also increases.

Reflection

The topology of CS model is like this:



At first, I just simply imitate this topology, so the structure I draw is like this:

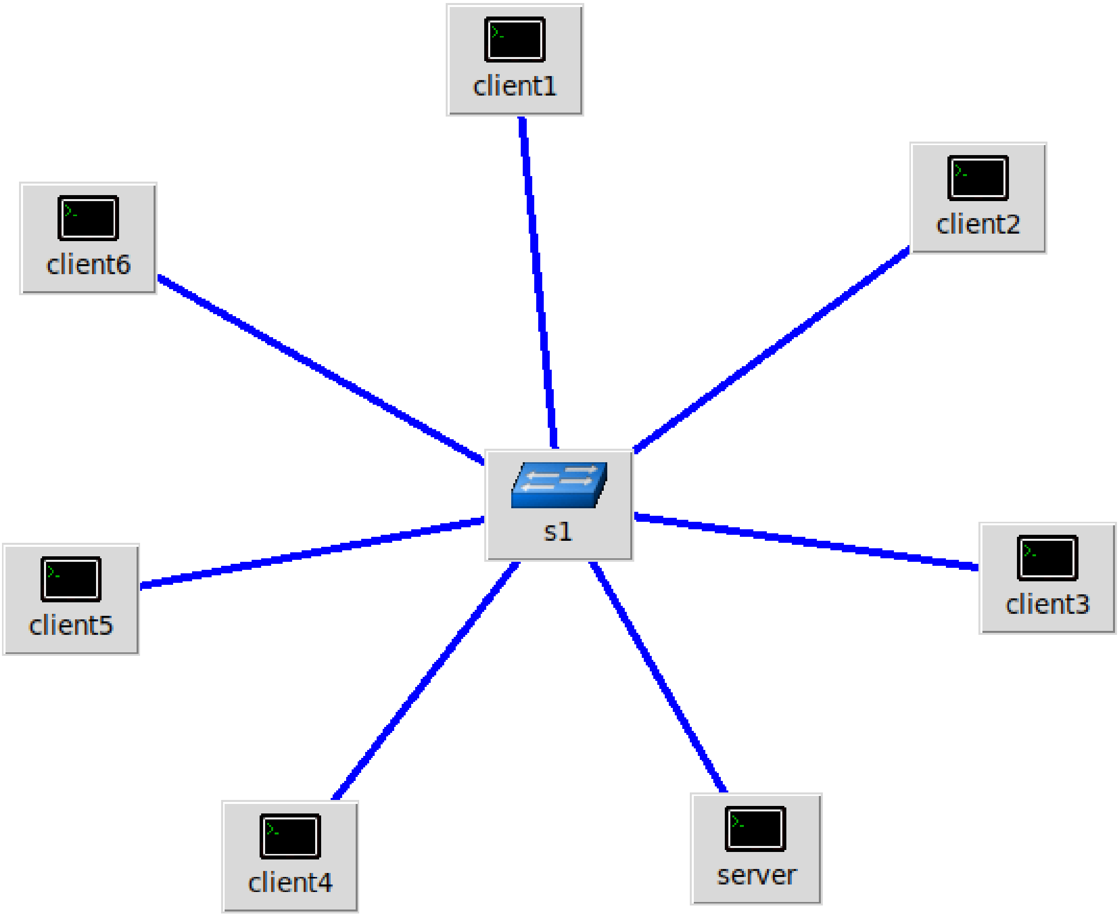


The problem of this topology is that every host cannot `ping` each other. So, I add these commands in the `mininet.py`:

```
    s1.cmd('ovs-ofctl add-flow s1 in_port=2,actions=output:1')
    s1.cmd('ovs-ofctl add-flow s1 in_port=1,actions=output:2')
    s2.cmd('ovs-ofctl add-flow s2 in_port=2,actions=output:1')
    s2.cmd('ovs-ofctl add-flow s2 in_port=1,actions=output:2')
    s3.cmd('ovs-ofctl add-flow s3 in_port=2,actions=output:1')
    s3.cmd('ovs-ofctl add-flow s3 in_port=1,actions=output:2')
    s4.cmd('ovs-ofctl add-flow s4 in_port=2,actions=output:1')
    s4.cmd('ovs-ofctl add-flow s4 in_port=1,actions=output:2')
    s5.cmd('ovs-ofctl add-flow s5 in_port=2,actions=output:1')
    s5.cmd('ovs-ofctl add-flow s5 in_port=1,actions=output:2')
    s6.cmd('ovs-ofctl add-flow s6 in_port=2,actions=output:1')
    s6.cmd('ovs-ofctl add-flow s6 in_port=1,actions=output:2')
```

So that the server can ping other hosts, but hosts themselves cannot ping each other.

After I ask the TA, I know a better way to construct the topology:



In this way, server and clients can ping each other natrually.

The reason is that different switches connect different network. So, in the first topology, I should configure the server as a router with 6 interfaces to different network range. But in the second topology, all clients and server are in the same network range, so they can ping each other naturally.