Table Of Contents

1. Solution

1.1 Approach towards the problem

Persistent: A single time connection to the socket for all the file transfers for a particular client
Non-persistent: For each new file transfer, a new connection is made.

TCP

Non Persistent Server Side	Persistent Server Side

`client_socket, address = server_socket.accept()`
`server_socket.settimeout(None)`
`print(f"connection to {address} has been established")`
 	
`file_indices= client_socket.recv(1024)`
`fileSend(client_socket,file_indices)`
`print("Finished Sending")`
 	
`client_socket.close()`
`print(f"connection with {address} is closed")`
	`client_socket, address = server_socket.accept()`
	`client_socket.settimeout(10)`
	`print(f"connection to {address} has been established")`
	
	`file_indices = client_socket.recv(32)`
	`while file_indices:`
	`  print(f"Client Requested to send {file_indices}")`
	`  fileSend(client_socket,file_indices)`
	
	`  file_indices = client_socket.recv(1024)`
	`  print(file_indices)`
	
	`print("Finished Sending all files")`
	`client_socket.close()`
	`print("Connection Closed")`

Non Persistent Client Side	Persistent Client Side

`file_string = "1 2 3 4 5"`
`filenames= file_string.strip().split()`
 	
`for file_index in filenames:`
 	
`  client_socket = socket.socket(socket.AF_INET,`
`  socket.SOCK_STREAM)`
 	
`  client_socket.connect((IP_ADDRESS,PORT))`
`  print(f"Requesting File with index: {file_index}")`
 	
`  client_socket.send(bytes(file_index,'utf-8'))`
`  fileReceive(client_socket,file_index)`
`  client_socket.close()`
	`file_string = "1 2 3 4 5"`
	`filenames= file_string.strip().split()`
	
	`client_socket = .socket(socket.AF_INET,socket.SOCK_STREAM)`
	`client_socket.connect((IP_ADDRESS,PORT))`
	`connection_setup_end = time.time()`
	
	`for file_index in filenames:`
	
	`  client_socket.send(bytes(file_index,'utf-8'))`
	`  fileReceive(client_socket,file_index)`
	
	`client_socket.close()`

Non Persistent	Persistent
Server Side: The server is programmed to send one complete file. After sending one file, it closes the	**Server Side:** It sends the multiple files, until there is no new request from the client side. (using while

connection.	loop).
Client Side: For receiving each file, a new socket connection is created (socket.socket()) and the respective socket is closed when the file is received completely.	**Client Side: T**he connection to the socket is made only a single time and the connection is terminated when all the files are received.

1.2 Experiment Results:

Experiments were performed five times to ensure the accuracy and reliability of the results. And the average of the experimental results has been considered for the analysis.

1.2.1 Persistent Connection:

All the times are in ms, and all the throughputs are in Mb/s.

TCP:

Exp. No	One time connection time	Total Download Time	Aggregate Throughput
1	0.25177	753.6006	17.23379
2	0.29802	761.39164	17.05744
3	0.2265	804.24809	16.14849
4	0.25082	741.32347	17.5192
5	0.24009	761.15537	17.06274
Average	**0.25344**	**764.34383**	**17.00433**

	Bleak House		Don Quixote		Les Miserables		Middle March		War and Peace	
Exp. No.	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	96.00139	20.84552	122.25032	19.55703	154.04797	21.87484	93.67013	19.92081	155.80249	21.5631
2	96.23599	20.7947	111.42206	21.45763	175.10056	19.2448	93.22929	20.01501	157.05252	21.39147
3	96.0238	20.84066	126.5893	18.88669	180.07827	18.71284	110.37278	16.90621	157.83763	21.28506
4	92.67759	21.59313	123.78049	19.31527	152.86565	22.04403	93.31083	19.99752	148.72599	22.58909
5	107.32532	18.6461	114.4073	20.89773	157.67932	21.37107	95.83426	19.47096	155.26414	21.63786
Average	**97.65282**	**20.54402**	**119.68989**	**20.02287**	**163.95435**	**20.64952**	**97.28346**	**19.2621**	**154.93655**	**21.69332**

UDP:

Exp. No.	Total Download Time	Aggregate throughput
1	2516.58583	3.3015
2	2701.94627	4.45057
3	2609.52926	4.52515
4	2586.231	5.30129
5	2787.0977	3.82939
Average	**2640.278012**	**4.28158**

Exp. No.	Anna Kareina		Bleak House		Don Quixote		Les Miserables		Middle March	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	136.93881	0.98707	325.11687	0.74331	529.47903	0.73491	690.11784	0.63507	834.93328	0.20114
2	121.98997	1.67883	346.06004	1.00607	516.27064	0.93619	788.4624	0.58703	929.16322	0.24245
3	188.49444	2.71626	366.53018	0.65933	518.71634	0.52906	686.37443	0.39386	849.41387	0.22664
4	169.85965	2.26672	322.19839	0.82632	462.98409	0.87585	747.85042	0.94753	883.33845	0.38487
5	162.64415	1.7125	365.2246	0.78505	556.67353	0.38997	787.36162	0.46299	915.1938	0.47888
Average	**155.9854**	**1.87228**	**345.02602**	**0.80402**	**516.82473**	**0.6932**	**740.03334**	**0.6053**	**882.40852**	**0.3068**

1.2.2 Non Persistent:

TCP

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Bleak House		Don Quixote		Les Miserables		Middle March		War and Peace	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	92.02051	21.74731	121.11449	19.74044	147.06564	22.91341	91.36796	20.42275	170.34745	19.72195
2	96.29726	20.78147	111.34243	21.47297	166.6472	20.22101	82.72862	22.55549	197.21627	17.03502
3	79.32234	25.22869	115.23366	20.74787	164.96468	20.42725	83.18472	22.43182	207.36837	16.20104
4	98.60516	20.29507	100.85392	23.7061	161.50331	20.86505	79.38671	23.505	219.97547	15.27254
5	93.54353	21.39324	136.92141	17.4615	165.17973	20.40066	80.35111	23.22289	205.17683	16.37409
Average	**91.95776**	**21.88916**	**117.09318**	**20.62578**	**161.07211**	**20.96548**	**83.40382**	**22.42759**	**200.01688**	**16.92093**

UDP

Exp. No.	Anna Kareina		Bleak House		Don Quixote		Les Miserables		Middle March	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	120.41593	2.82328	310.80317	0.46126	518.56995	0.75037	719.22016	0.44421	846.60888	0.27094
2	207.2506	2.74713	405.69258	0.95915	633.92925	0.78828	808.89106	0.66335	935.42457	0.36344
3	175.09913	1.38016	328.45783	1.03504	504.89616	0.48675	690.51194	0.6703	852.65875	0.11049
4	162.17637	1.21231	353.97482	0.96043	493.22772	0.68927	633.44097	0.5755	852.38934	0.28351
5	149.06669	1.20902	332.66616	1.02195	543.13946	0.62593	778.03421	0.68966	901.99232	0.10444
Average	**162.80174**	**1.87438**	**346.31891**	**0.88757**	**538.75251**	**0.66812**	**726.01967**	**0.6086**	**877.81477**	**0.22656**

1.3 Observation:

1. In Both TCP and UDP, non-persistent seems to outperform persistent connections.
2. The results are a bit contradictory to what is expected. Theoretically, the persistent connection needs to be faster than the non persistent one. But, the villain here is the throughput. The throughput is only around 20 Mb/s which is very much slower than the local drive. Therefore in slower connections, the memory-based non-persistent connections would be faster than the non-persistent connections.[1]

[1] "PERSISTENT message have much slower performance than ...." 4 Jul. 2012, https://stackoverflow.com/questions/11303935/persistent-message-have-much-slower-performance-than-non-persistent-message. Accessed 23 Nov. 2020.

2. Solution

2.1 Fork Model

```
'''
Reference:
https://github.com/gulshan-mittal/Socket-Programming/blob/master

'''
import socket
import time
import os
import argparse

#Argument Parser Section
ap =argparse.ArgumentParser()
ap.add_argument("-i","--index",help ="index to the dictionary")
ap.add_argument("-p","--persistent",help ="1 for persistent 0 for non persistent")
args = vars(ap.parse_args())

#variables

HOST = socket.gethostname()
PORT = 12345
BUFFER = 1024

file_name_list = os.listdir('./text_files')
index_to_names = {}
for key,name in enumerate(file_name_list):
   index_to_names[key]=name



server_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server_socket.bind((HOST,PORT))
server_socket.listen(5)


def fileSend(client_socket,file_indices):

   print("im here")

   try:
       filename = index_to_names[int(file_indices.decode())]
       sendFile = open('./text_files/'+filename,'rb')
       client_socket.send(bytes("---File Present---"+filename,'utf-8'))


   except IOError:
       client_socket.send(bytes("---File Not Present---",'utf-8'))
       return

   ack= client_socket.recv(64)
   print(f"Started to send {filename}")
   read_data=sendFile.read(BUFFER)
```

```
    while (read_data):

            client_socket.send(read_data)
            ack = client_socket.recv(64)
            read_data=sendFile.read(BUFFER)

    sendFile.close()
    client_socket.send(bytes("EOF",'utf-8'))


def nonPersistentFork(client_socket,address):

    server_socket.settimeout(None)
    print(f"connection to {address} has been established")
    file_indices= client_socket.recv(1024)

    fileSend(client_socket,file_indices)


    print("Finished Sending")
    client_socket.close()
    print(f"connection with {address} is closed")

def nonPersistentConnection():
    client_socket, address = server_socket.accept()
    new_pid = os.fork()

    if new_pid==0:
        nonPersistentFork(client_socket,address)


def persistentFork(client_socket,address):
    client_socket.settimeout(10)
    print(f"connection to {address} has been established")

    file_indices = client_socket.recv(32)

    while file_indices:

        print(f"Client Requested to send {file_indices}")
        fileSend(client_socket,file_indices)

        file_indices = client_socket.recv(1024)
        print(file_indices)
    print("Finished Sending all files")
    client_socket.close()
    print("Connection Closed")


def persistentConnection():

    client_socket, address = server_socket.accept()
    new_pid = os.fork()

    if new_pid==0:
        persistentFork(client_socket,address)

print("Server listening....")
while True:

    start_time = time.time()
```

```
    #nonPersistentConnection()

    if args['persistent']=='1':
        print("persistent Connection")
        persistentConnection()

    elif args['persistent']=='0':
        print("Non Persistent Connection")
        nonPersistentConnection()
```

A new child process is started by using os.fork() which returns a pid value of 0 for child.

2.2 Thread Model:

```
'''
Reference:
https://github.com/gulshan-mittal/Socket-Programming/blob/master

'''
import socket
import time
import os
import argparse
import threading

#Argument Parser Section
ap =argparse.ArgumentParser()
ap.add_argument("-i","--index",help ="index to the dictionary")
ap.add_argument("-p","--persistent",help ="1 for persistent 0 for non persistent")
args = vars(ap.parse_args())

#variables

HOST = socket.gethostname()
PORT = 12345
BUFFER = 1024

file_name_list = os.listdir('./text_files')
index_to_names = {}
for key,name in enumerate(file_name_list):
    index_to_names[key]=name



server_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server_socket.bind((HOST,PORT))
server_socket.listen(5)


def fileSend(client_socket,file_indices):

    print("im here")
```

```
    try:
        filename = index_to_names[int(file_indices.decode())]
        sendFile = open('./text_files/'+filename,'rb')
        client_socket.send(bytes("---File Present---"+filename,'utf-8'))


    except IOError:
        client_socket.send(bytes("---File Not Present---",'utf-8'))
        return

    ack= client_socket.recv(64)
    print(f"Started to send {filename}")
    read_data=sendFile.read(BUFFER)
    while (read_data):

            client_socket.send(read_data)
            ack = client_socket.recv(64)
            read_data=sendFile.read(BUFFER)

    sendFile.close()
    client_socket.send(bytes("EOF",'utf-8'))


def nonPersistentThread(client_socket,address):

    server_socket.settimeout(None)
    print(f"connection to {address} has been established")
    file_indices= client_socket.recv(1024)

    fileSend(client_socket,file_indices)


    print("Finished Sending")
    client_socket.close()
    print(f"connection with {address} is closed")

def nonPersistentConnection():
    client_socket, address = server_socket.accept()
    threading._start_new_thread(nonPersistentThread,(client_socket,address))


def persistentThread(client_socket,address):
    client_socket.settimeout(10)
    print(f"connection to {address} has been established")

    file_indices = client_socket.recv(32)

    while file_indices:

        print(f"Client Requested to send {file_indices}")
        fileSend(client_socket,file_indices)

        file_indices = client_socket.recv(1024)
        print(file_indices)
    print("Finished Sending all files")
    client_socket.close()
    print("Connection Closed")


def persistentConnection():
```

```
    client_socket, address = server_socket.accept()
    threading._start_new_thread(persistentThread,(client_socket,address))


print("Server listening....")
while True:

    if args['persistent']=='1':
        print("persistent Connection")
        persistentConnection()

    elif args['persistent']=='0':
        print("Non Persistent Connection")
        nonPersistentConnection()
```

When a new connection comes, a new thread is started by using start_new_thread function from threading.

2.3 (C) Download Multiple files using concurrent servers

2.3.1 Fork Model Results for five files using persistent connection.

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No	One time connection time	Total Download Time	Aggregate Throughput
1	0.88406	1193.45975	10.88214
2	0.31853	816.32757	15.90954
3	0.2861	828.58396	15.67421
4	0.3345	814.75854	15.94018
5	0.28086	826.59674	15.71189
Average	**0.42081**	**895.94531**	**14.82359**

	Bleak House		Don Quixote		Les Miserables		Middle March		War and Peace	
Exp. No.	Dwld Time	Through put	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	174.8724	11.44377	245.10312	9.75448	220.03388	15.3148	184.16047	10.13239	219.71703	15.2905
2	113.87658	17.5734	115.04674	20.78158	177.99735	18.9316	81.94923	22.77001	194.21792	17.29801
3	104.05874	19.23144	135.40816	17.65664	162.28986	20.76393	94.2862	19.79065	194.50021	17.27291
4	118.84785	16.83833	115.60225	20.68172	171.58461	19.63915	89.33043	20.88857	190.51385	17.63433
5	101.54009	19.70846	131.18362	18.22524	188.37309	17.88883	92.20099	20.23823	169.69347	19.79796
Average	**122.63913**	**16.95908**	**148.46878**	**17.41993**	**184.05576**	**18.50766**	**108.38546**	**18.76397**	**193.7285**	**17.45874**

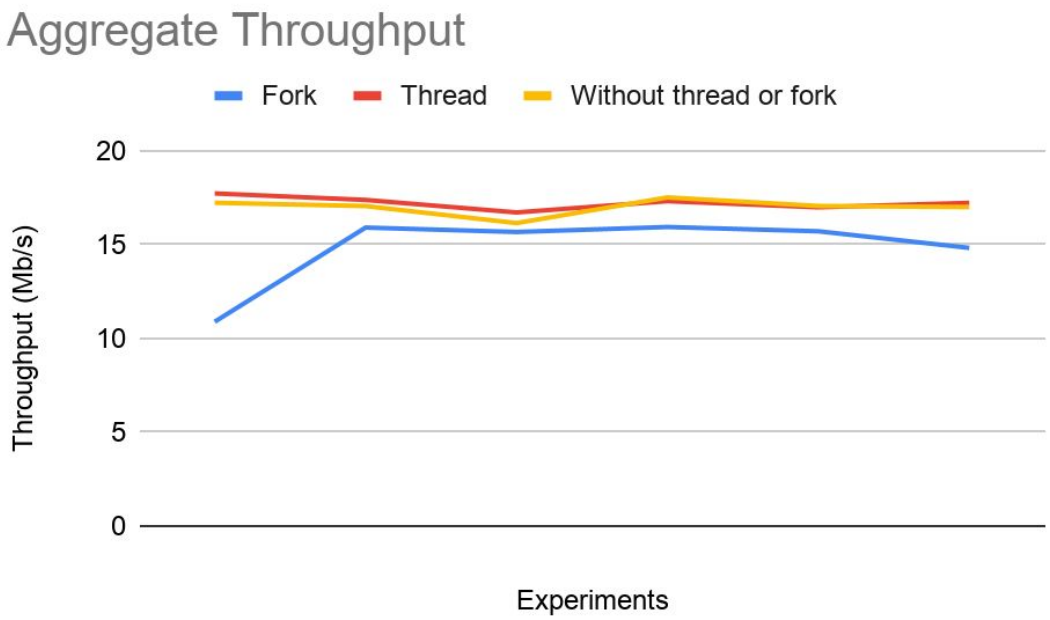
2.3.2 Thread Model Results for downloading five files using persistent connection.

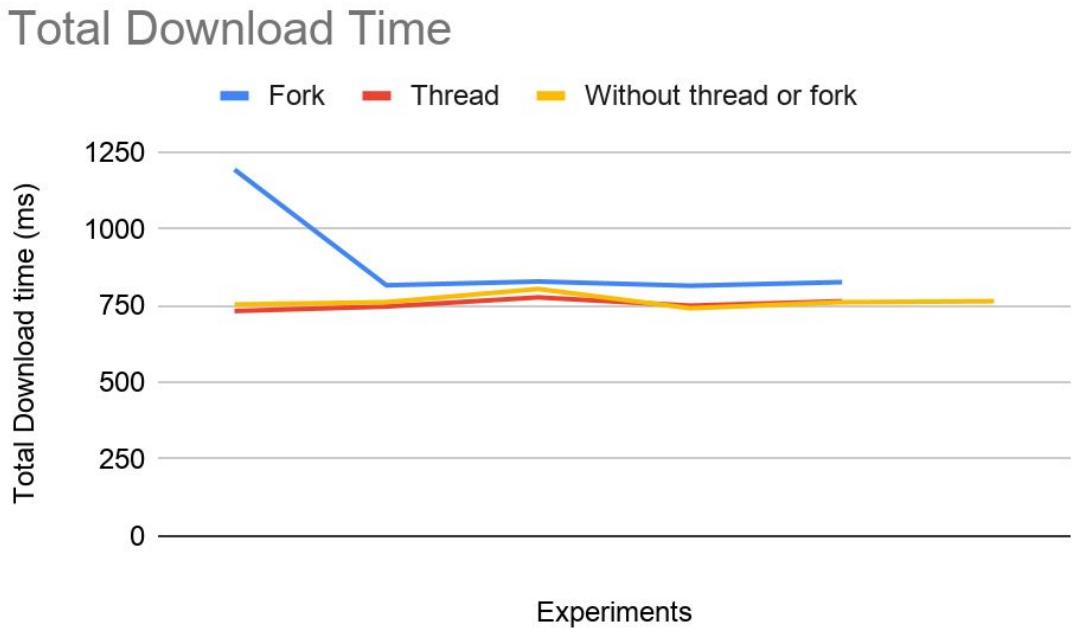
All the times are in ms, and all the throughputs are in Mb/s.

Exp. No	One time connection time	Total Download Time	Aggregate Throughput
1	0.42701	732.42283	17.7321
2	0.26202	747.06101	17.38465
3	0.66376	776.90148	16.71691
4	0.26369	749.90606	17.3187
5	0.58222	764.3168	16.99216
Average	**0.43974**	**754.12164**	**17.2289**

Exp. No.	Bleak House		Don Quixote		Les Miserables		Middle March		War and Peace	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	113.02876	17.70522	103.01328	23.20917	151.14903	22.29439	87.8787	21.23364	146.58713	22.91868
2	111.77468	17.90387	112.67424	21.21916	154.87194	21.75846	90.77716	20.55567	141.43896	23.75289
3	130.49865	15.33502	115.95368	20.61903	148.83161	22.64153	89.38813	20.87509	151.17717	22.22283
4	102.46849	19.5299	109.47084	21.84009	154.46019	21.81646	95.83116	19.47159	153.03802	21.95261
5	119.70568	16.71766	116.44459	20.53211	145.29204	23.19311	92.8731	20.09177	156.27265	21.49822
Average	**115.49525**	**17.43833**	**111.51133**	**21.48391**	**150.92096**	**22.34079**	**91.34965**	**20.44555**	**149.70279**	**22.46905**

2.3.3 Observations:





1. It is evident from the above two plots, thread based model has lower download time and higher aggregate throughput relative to that of fork based model. **Aggregate throughput of thread model is 13.96% higher than that of fork model.** From this experiment, we can conclude that thread model works best for our case.
2. When compared with individual throughputs and download time, thread model can be seen to outperform fork model.

3. When compared to the server without thread or fork, for based model performed poorly while thread based model was in par with that of without thread or fork model. It can be seen that, the values are almost similar. However, the **aggregate throughput of thread model is 1.303% higher than that of without thread or fork model**.
4. From the observation, we can see that thread based concurrent servers are more efficient than non concurrent servers and fork based concurrent servers. This observation can be related to the remarks to the experiments in the first question. Since the throughput is only in the range of 20 Mb/s, it is slower than the local drive speed. Therefore, by handling more threads, more memory would be consumed, but the throughput would be higher. And when comparing thread vs fork, thread model can share parents resources while fork cannot. Fork is treated as a child while thread is considered as a sibling.[2]
5. Latency by definition means the amount of time it takes for data to travel from source to destination, usually measured in milliseconds. Generally speaking, the lower the latency the better. Overhead is the excess resources required to perform a specific task such as transfer data.
6. Lower latency is obtained in case of thread based model. But, it used excess resources or greater overhead due to reason stated in point 4. Though, the fork based model has lower overhead, its latency is higher rendering it less efficient (it can be reasoned out by the sibling child analogy).

[2] "Forking vs Threading - Stack Overflow." 22 Jun. 2016, https://stackoverflow.com/questions/16354460/forking-vs-threading. Accessed 24 Nov. 2020.

2.4 (D) Five clients experiment

2.4.1 Thread based server (Non persistent):

In this set of experiments each client downloads a file and each client downloads a different file.

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput
1	1592.84449	40.68551
2	2584.49745	24.8723
3	1871.36007	34.55497
4	1835.55221	34.91194
5	2671.40198	24.06097
Average	**2111.13124**	**31.817138**

Exp. No.	Client 1		Client 2		Client 3		Client 4		Client 5	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	256.79851	7.79288	263.59916	7.07887	262.68506	9.10159	435.15897	7.74378	374.60279	8.96839
2	458.06503	4.36881	429.46434	4.34491	564.27598	4.23703	534.17444	6.30838	598.51766	5.61317
3	319.15331	6.27034	277.23789	6.73063	383.60143	6.23265	391.65497	8.57792	499.71247	6.74343
4	301.94473	6.6277	312.99615	5.96169	342.64421	6.97765	453.26543	7.43444	424.70169	7.91046
5	457.20148	4.37706	395.96486	4.7125	513.42154	4.65671	651.57604	5.17173	653.23806	5.14297
Average	**358.63261**	**5.88736**	**335.85248**	**5.76572**	**413.32564**	**6.24113**	**493.16597**	**7.04725**	**510.15453**	**6.87568**

2.4.2 Fork Based Server (Non Persistent):

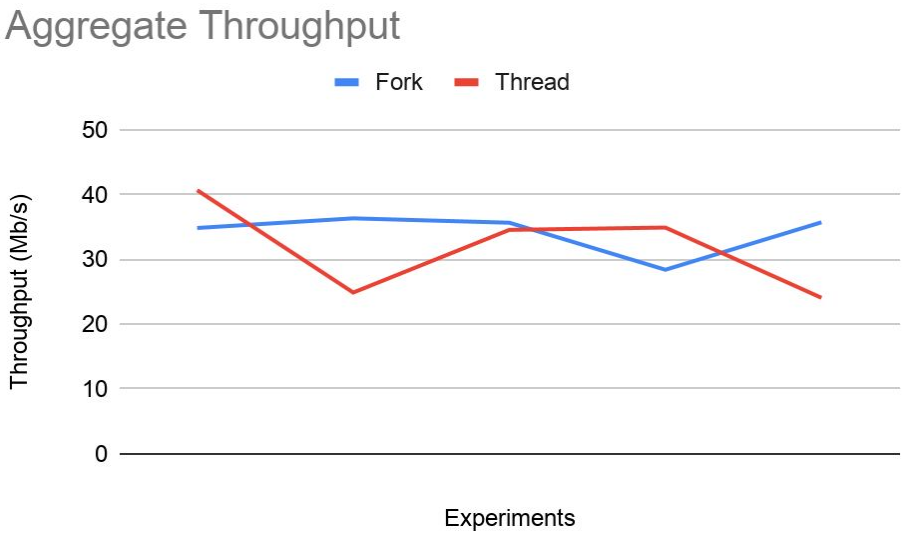
All the times are in ms, and all the throughputs are in Mb/s.

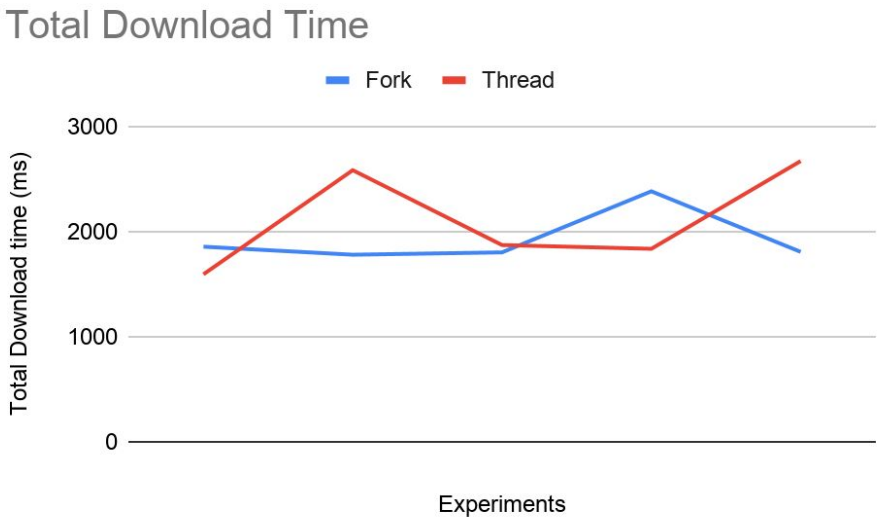
Exp. No.	Total Download Time	Aggregate throughput
1	1855.67736	34.84639
2	1779.23274	36.33543
3	1801.62334	35.65981
4	2383.29863	28.39505
5	1806.6225	35.72324
Average	**1925.290914**	**34.191984**

Exp. No.	Client 1		Client 2		Client 3		Client 4		Client 5	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	272.69864	6.84266	311.80429	6.41813	324.37825	7.37057	474.36047	7.10383	472.43571	7.1112
2	269.96803	6.91187	264.36281	7.5699	380.61047	6.28163	431.53405	7.80883	432.75738	7.7632
3	276.03889	6.75986	299.66211	6.67818	352.11682	6.78994	456.30932	7.38485	417.4962	8.04698
4	344.50674	5.4164	264.36281	7.5699	556.55217	4.29583	563.92932	5.97553	653.94759	5.13739
5	264.31537	7.05969	304.10194	6.58068	327.13628	7.30843	450.58441	7.47868	460.4845	7.29576

Average	**285.50553**	**6.5981**	**288.85879**	**6.96336**	**388.1588**	**6.40928**	**475.34351**	**7.15034**	**487.42428**	**7.07091**

2.4.3 Analysis:





1. There is greater fluctuation in the results when done multiple times. By taking an average of the values, thread based model perform well. The **aggregate throughput of the fork model is higher than that of the fork model by 6.9456%.**
2. Overall the aggregate throughput has increased. For single client thread based server, the average aggregate throughput was **17.2289 Mb/s.** For five clients, the average aggregate throughput is **31.817 Mb/s.** Which is a **45.85%** increase in aggregate throughput. Similarly if we observe for forkbased model, there is an increase of **56.64%** aggregate throughput.
3. However,throughput per client has decreased. For a single connection it was **17.2289 Mb/s** for thread based model. While for the five client scenario, the throughput per client reduced to **6.363 Mb/s** for thread based model. The reduction in throughput is of **63.056%.** Similarly for fork model, the reduction in throughput per

client is **53.868%**. This is expected since the resource is split among five clients. However, the rise in the aggregate throughput was unexpected. It may be due to more efficient utilization of memory resources.

4. From the experiments we can conclude that fork based model is preferred over thread based models for concurrent servers. When there are multiple clients, fork model tends to outperform thread model. The reason can be the resource utilization as five clients will need much more resources in thread based model than that for fork based model.

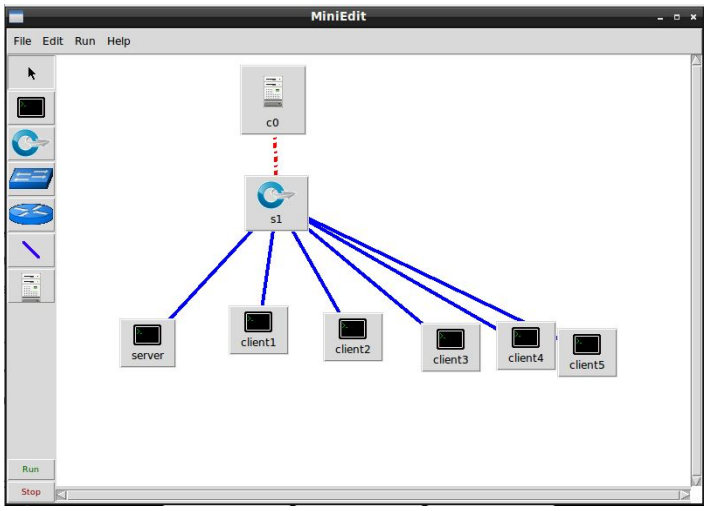
3. Solution:

3.1 Steps at the mininet side:

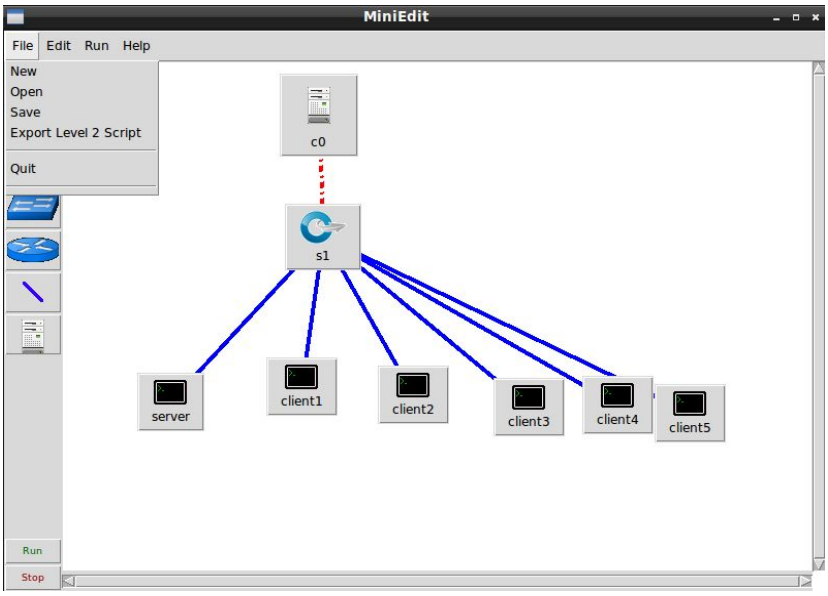
MiniEdit GUI of mininet is used for creating the required topology. MiniEdit can be accessed from the mininet terminal by using the command:

```
root@mininet-vm:~# sudo python mininet/examples/miniedit.py
```

Created the required topology:



Export the python script for the custom topology using File->export level2 script:



Run the custom topology python script in mininet:

```
$sudo python custom_topology.py
```

Ping Test Whether All hosts and links are working properly

```
mininet> pingall
*** Ping: testing ping reachability
client4 -> client1 server client5 client2 client3
client1 -> client4 server client5 client2 client3
server -> client4 client1 client5 client2 client3
client5 -> client4 client1 server client2 client3
client2 -> client4 client1 server client5 client3
client3 -> client4 client1 server client5 client2
*** Results: 0% dropped (30/30 received)
mininet>
```

Find the IP address of the server, so that the server program can listen to that particular IP address as well as clients can listen to that particular IP address.

```
mininet> dump
<Host client4: client4-eth0:10.0.0.5 pid=29822>
<Host client1: client1-eth0:10.0.0.2 pid=29824>
<Host server: server-eth0:10.0.0.1 pid=29826>
<Host client5: client5-eth0:10.0.0.6 pid=29828>
<Host client2: client2-eth0:10.0.0.3 pid=29830>
<Host client3: client3-eth0:10.0.0.4 pid=29832>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:N
one,s1-eth5:None,s1-eth6:None pid=29817>
<Controller c0: 127.0.0.1:6633 pid=29809>
mininet>
```

IP address of the server is 10.0.0.1.

Running the server and Client Programs:

1. Either can run inside mininet cli
   $ server python tcp_server.py
   $ client1 python tcp_client.py
2. Or use xterm to get individual terminal and run the code.
   $ xterm server
   $xterm client1
3. Using python mininet CLI. Edit inside the custom topology to run the command. Add the following lines to the custom topology python script after topology creation.

   server.cmd("python tcp_server.py &")
   client1.cmd("python tcp_client.py &")
4. Commands in 1 can be run all together at once in mininet by saving the commands inside a .sh file and running $>mininet source test.sh

   Contents of test.sh
   Server python tcp_server.py
   Client1 python tcp_client.py

3.2 (E) Single Topology (i)

Each Client Downloaded Different Files:

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput
1	1418.0839	45.6211
2	1430.99404	45.13243
3	1459.55848	44.21685
4	1446.95734	44.56489
5	1441.00571	44.78637
Average	**1439.319894**	**44.864328**

Exp. No.	Client 1		Client 2		Client 3		Client 4		Client 5	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	212.70037	8.77283	237.66398	8.70169	269.86718	8.85937	352.06628	9.57142	345.78609	9.71579
2	223.25516	8.35808	239.63237	8.63022	268.14914	8.91613	349.43986	9.64336	350.51751	9.58464
3	227.76699	8.19252	248.46005	8.32359	273.76747	8.73315	354.72798	9.4996	354.83599	9.46799
4	227.87571	8.18861	246.60182	8.38631	275.07591	8.69161	348.37174	9.67293	349.03216	9.62543
5	223.15001	8.36202	244.93051	8.44353	274.64104	8.70537	351.92537	9.5463	346.35878	9.72915
Average	**222.94965**	**8.37481**	**243.45775**	**8.49707**	**272.30015**	**8.78113**	**351.30625**	**9.58672**	**349.30611**	**9.6246**

3.2.1 Observations

1. The Individual throughput of client has increased significantly.
2. The aggregate throughput of Non Mininet was **34.19 Mb/s** while the mininet version it is **44.86 Mb/s**. There is an **increase of 23.79% in the throughput**.
3. The increase may be due to the increase in speed in the mininet OS due to the lightness of the system. Or may be the efficiency of mininet in implementing the network topology.

3.3 (F) Single Topology (ii)

All the times are in ms, and all the throughputs are in Mb/s.

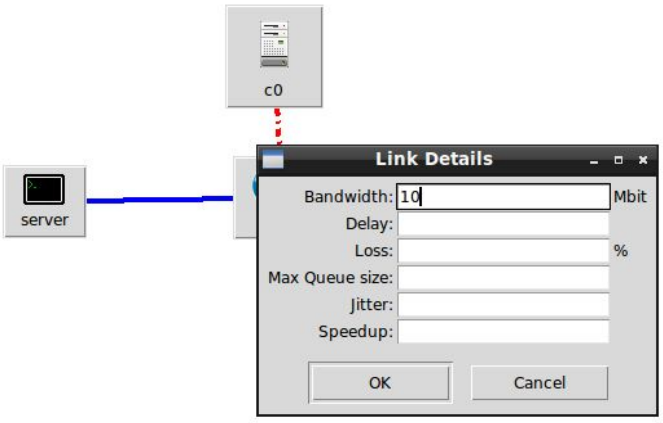
Exp. No.	Total Download Time	Aggregate throughput
1	1414.71314	59.59546
2	1505.40208	55.968
3	1580.10172	53.3278
4	1579.81204	53.32803
5	1574.37468	53.51001
Average	**1530.880732**	**55.14586**

Exp. No.	Client 1		Client 2		Client 3		Client 4		Client 5	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	269.88339	12.48604	278.7869	12.08728	290.37452	11.60493	290.81845	11.58721	284.84988	11.83
2	298.10905	11.30383	303.06029	11.11916	303.67947	11.09649	296.24486	11.37497	304.30841	11.07355
3	316.81561	10.63639	307.71112	10.9511	322.06154	10.46314	315.32359	10.68672	318.18986	10.59045
4	312.43896	10.78539	316.48278	10.64758	314.77308	10.70541	317.99722	10.59687	318.12	10.59278
5	315.51838	10.68012	315.78207	10.6712	313.91025	10.73484	314.24999	10.72323	314.91399	10.70062
Average	**302.55308**	**11.17835**	**304.36463**	**11.09526**	**308.95977**	**10.92096**	**306.92682**	**10.9938**	**308.07643**	**10.95748**

3.3.1 Observations

1. When all the clients tried to download the same file, the throughput can be seen increased.
2. When comparing the aggregate throughput with that of downloading different files, there is an **increase of 18.64%.**
3. It can be due to the caching in the server memory. Since the same file is downloaded, the file is already cached, the server need not cache it always when a new client requests for that particular file. Which increases faster processing time at the server side.

3.4 (G) Linear Topology (i) Bandwidth



All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Bandwidth 10 Mbps		Bandwidth 100 Mbps		Bandwidth 1000 Mbps	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	2896.63935	9.30672	296.25845	90.99555	91.36248	295.06862
2	2886.29198	9.34008	297.1251	90.73013	89.81538	300.15127
3	2906.60906	9.27479	297.92547	90.48639	89.04457	302.74951
4	2925.19307	9.21587	297.89639	90.49522	126.75905	212.67278
5	2894.12284	9.31481	300.20213	89.80016	96.99965	277.9206
Average	**2901.77126**	**9.29045**	**297.88151**	**90.50149**	**98.79623**	**277.71256**

3.4.1 Observations

- As bandwidth increased, the throughput also increased as expected.
- When the bandwidth was increased to 1 Gbps, the maximum throughput obtained was 302 Mbps. It may be due to the resource limitations at the server side and client side and not a limitation of link layer.
- The maximum throughput is obtained for 1Gbps link as expected.

3.5 (H) Linear Topology (ii) Delay

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Delay 1ms		Delay 2ms		Delay 5ms		Delay 10ms	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	17192.52777	1.56802	31337.39901	0.86026	73232.07688	0.36812	139452.6715	0.19331
2	17207.87168	1.56662	31873.87347	0.84578	72805.12238	0.37028	138987.4835	0.19396
3	17189.51797	1.56829	31819.10467	0.84723	72689.40592	0.37087	138899.2138	0.19408
4	17249.02821	1.56288	31814.0595	0.84737	71912.6873	0.37487	139199.873	0.19367
5	17212.00371	1.56624	31934.44014	0.84417	72853.58572	0.37003	139724.4253	0.19294
Average	**17210.18987**	**1.56641**	**31755.77536**	**0.84896**	**72698.57564**	**0.37083**	**139252.7334**	**0.19359**

3.5.1 Observations

- With Increasing delay, the throughput can be seen to decrease.
- The relation between the delay and throughput is approximately linear.
- It is as expected since delay in the links leads to decrease in the throughput.

3.6 (I) Linear Topology (ii) Loss

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	1% loss		2% loss		5% loss	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	32269.04058	0.83542	31337.39901	0.86026	0	0
2	33295.05229	0.80968	31873.87347	0.84578	0	0
3	26791.19539	1.00623	31819.10467	0.84723	0	0
4	26304.37398	1.02486	31814.0595	0.84737	0	0
5	30804.90851	0.87513	31934.44014	0.84417	0	0
Average	**29892.91415**	**0.91026**	**31755.77536**	**0.84896**	**0**	**0**

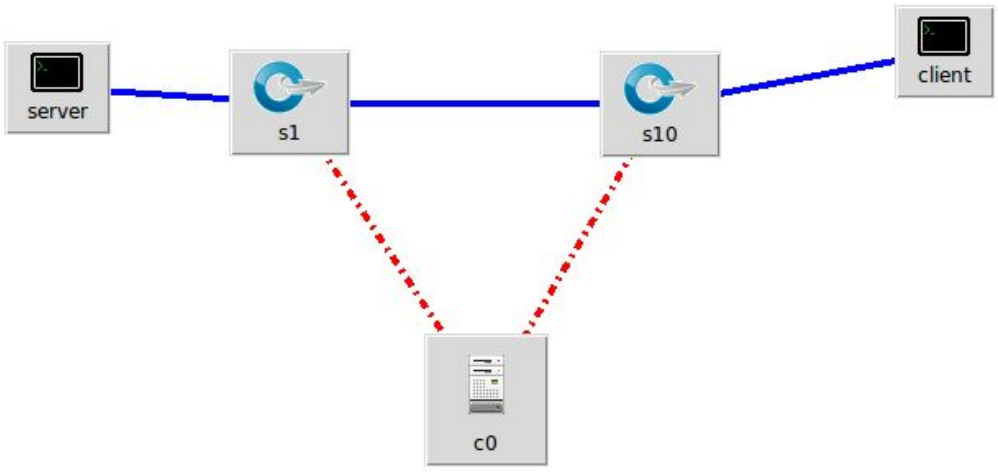
3.6.1 Observations

- With Increasing loss, the throughput can be seen to decrease.
- It proves the theory that delays in the links can lead to decrease in the link speeds in TCP connections.

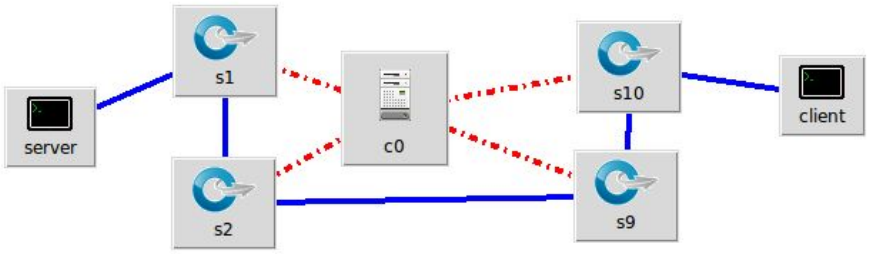
3.7 (J) Linear Topology (ii) Hops

The required topology is created using miniedit and it is exported as python mininet cli script. And it is simulated inside the mininet for the results. The codes for the same is shared along with the report.

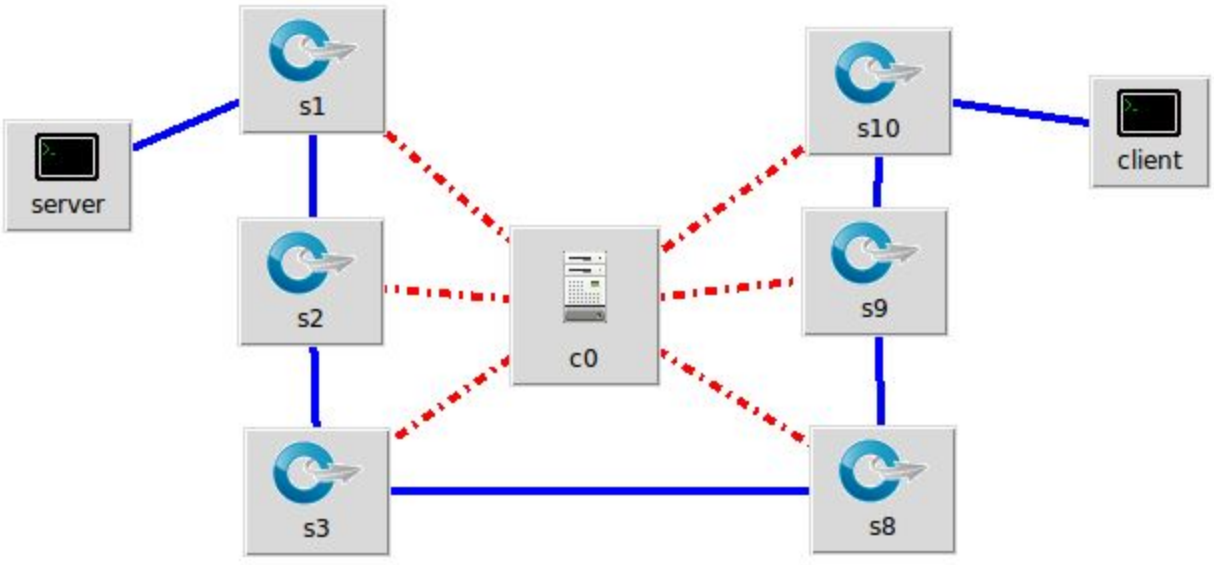
With 2 switches - MiniEdit GUI



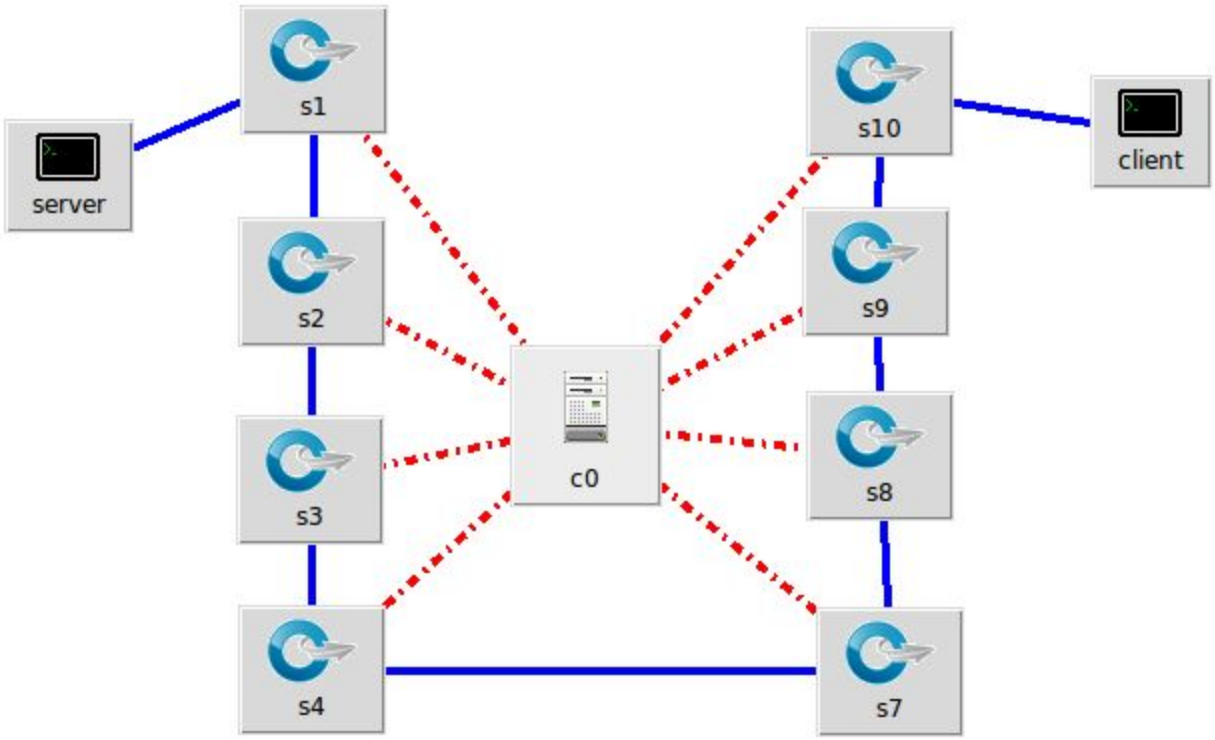
With 4 switches - MiniEdit GUI



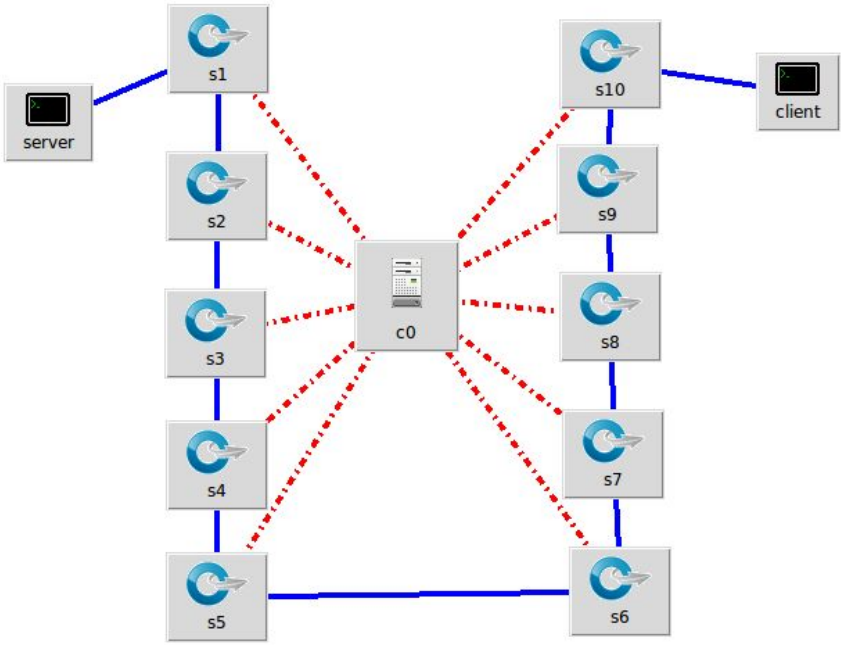
With 6 switches



With 8 switches



With 10 Switches - MiniEdit GUI



All the times are in ms, and all the throughputs are in Mb/s.

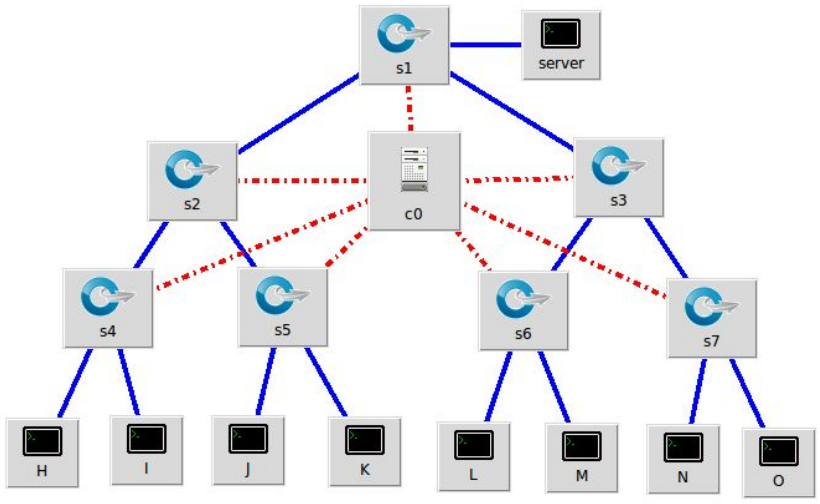
Exp. No.	2 Switches		4 Switches		6 Switches		8 Switches		10 Switches	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	79.39386	339.55017	86.12943	312.99639	100.70753	267.68802	113.41643	237.69219	123.63434	218.04784
2	71.9862	374.49123	83.75525	321.86876	93.9889	286.82325	109.63011	245.90143	127.88463	210.80094
3	128.69596	209.47199	84.7888	317.9453	92.93342	290.0808	109.62486	245.91319	155.25079	173.64292
4	70.58263	381.93817	83.11415	324.35152	92.27061	292.16453	103.60193	260.20943	120.03636	224.58361
5	90.56902	297.65366	81.22563	331.89276	96.3273	279.86042	103.71304	259.93068	119.99679	224.65768
Average	**88.24553**	**320.62104**	**83.80265**	**321.81095**	**95.24555**	**283.3234**	**107.99727**	**249.92938**	**129.36058**	**210.3466**

3.6.1 Observations

- With increase in number of switches, the throughput decreases and the time taken to download the file increases.
- It is evident that the number of switches is more in a network, there would be various delays coming from the switches also which results in reduced performance.

4. Solution:

Custom Topology in MiniEdit GUI



```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

   net = Mininet( topo=None,
                  build=False,
                  ipBase='10.0.0.0/8')

   info( '*** Adding controller\n' )
   c0=net.addController(name='c0',
                     controller=Controller,
                     protocol='tcp',
                     port=6633)

   info( '*** Add switches\n')
   s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
   s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
   s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
   s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
   s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
   s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
   s7 = net.addSwitch('s7', cls=OVSKernelSwitch)

   info( '*** Add hosts\n')
   L = net.addHost('L', cls=Host, ip='10.0.0.6', defaultRoute=None)
   K = net.addHost('K', cls=Host, ip='10.0.0.5', defaultRoute=None)
   server = net.addHost('server', cls=Host, ip='10.0.0.1', defaultRoute=None)
   O = net.addHost('O', cls=Host, ip='10.0.0.9', defaultRoute=None)
   N = net.addHost('N', cls=Host, ip='10.0.0.8', defaultRoute=None)
   J = net.addHost('J', cls=Host, ip='10.0.0.4', defaultRoute=None)
   H = net.addHost('H', cls=Host, ip='10.0.0.2', defaultRoute=None)
   I = net.addHost('I', cls=Host, ip='10.0.0.3', defaultRoute=None)
   M = net.addHost('M', cls=Host, ip='10.0.0.7', defaultRoute=None)

   info( '*** Add links\n')
   s1s2 = {'bw':40}
   net.addLink(s1, s2, cls=TCLink , **s1s2)
   s2s4 = {'bw':20}
   net.addLink(s2, s4, cls=TCLink , **s2s4)
   s2s5 = {'bw':20}
   net.addLink(s2, s5, cls=TCLink , **s2s5)
   s4H = {'bw':10}
   net.addLink(s4, H, cls=TCLink , **s4H)
```

```
    s4I = {'bw':10}
    net.addLink(s4, I, cls=TCLink , **s4I)
    s5J = {'bw':10}
    net.addLink(s5, J, cls=TCLink , **s5J)
    s5K = {'bw':10}
    net.addLink(s5, K, cls=TCLink , **s5K)
    s6L = {'bw':10}
    net.addLink(s6, L, cls=TCLink , **s6L)
    s6M = {'bw':10}
    net.addLink(s6, M, cls=TCLink , **s6M)
    s7N = {'bw':10}
    net.addLink(s7, N, cls=TCLink , **s7N)
    s7O = {'bw':10}
    net.addLink(s7, O, cls=TCLink , **s7O)
    s1server = {'bw':40}
    net.addLink(s1, server, cls=TCLink , **s1server)
    s1s3 = {'bw':40}
    net.addLink(s1, s3, cls=TCLink , **s1s3)
    s3s6 = {'bw':20}
    net.addLink(s3, s6, cls=TCLink , **s3s6)
    s3s7 = {'bw':20}
    net.addLink(s3, s7, cls=TCLink , **s3s7)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s4').start([c0])
    net.get('s1').start([c0])
    net.get('s5').start([c0])
    net.get('s6').start([c0])
    net.get('s3').start([c0])
    net.get('s2').start([c0])
    net.get('s7').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

In mininet, all the clients are started simultaneously using scripts named “grace_a.sh” and “grace_b.sh”

```
server python3.8 tcp_server_sc_thread.py -p 0 &
H python3.8 tcp_client_new.py -p 0 -i "1" &
I python3.8 tcp_client_new.py -p 0 -i "2" &
J python3.8 tcp_client_new.py -p 0 -i "3" &
K python3.8 tcp_client_new.py -p 0 -i "4"
```

```
server python3.8 tcp_server_sc_thread.py -p 0 &
H python3.8 tcp_client_new.py -p 0 -i "1" &
K python3.8 tcp_client_new.py -p 0 -i "2" &
M python3.8 tcp_client_new.py -p 0 -i "3" &
N python3.8 tcp_client_new.py -p 0 -i "4"
```

4.1 (K) Custom Topology i) Varying Bandwidth

4.1.1 Clients on HIJK

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput
1	8662.65129	35.90406
2	8678.82443	35.83868
3	8651.58367	35.93587
4	8664.79588	35.84795
5	8650.92612	35.9093
Average	**8661.756278**	**35.887172**

Exp. No.	H		I		J		K	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	1620.20206	9.21359	1833.81867	9.02196	2162.06002	8.84657	3046.57054	8.82194
2	1839.95891	8.99185	2146.7011	8.90987	3062.79397	8.77521	1629.37045	9.16175
3	2155.27177	8.87444	3038.00178	8.84683	1620.07999	9.21429	1838.23013	9.00031
4	3036.66759	8.85071	1639.41884	9.10559	1842.09514	8.98142	2146.61431	8.91023
5	1623.17896	9.19669	1849.73216	8.94434	2153.2805	8.88264	3024.7345	8.88563
Average	**2055.05586**	**9.02546**	**2101.53451**	**8.96572**	**2168.06192**	**8.94003**	**2337.10399**	**8.95597**

4.1.2 Clients on HKMN

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput

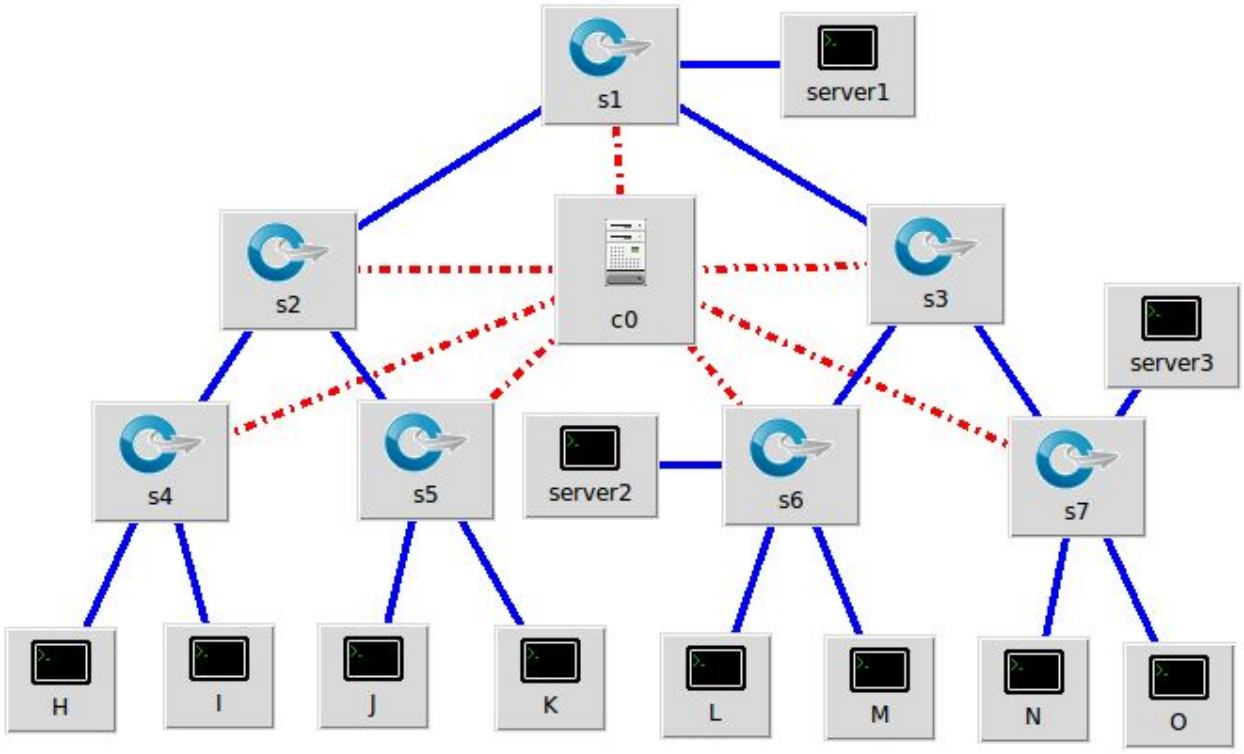
1	8678.19428	35.78743
2	8729.28167	35.60827
3	8706.2831	35.70675
4	8691.77984	35.82958
5	8656.31247	35.9483
Average	**8692.370272**	**35.776066**

Exp. No.	H		K		M		N	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	1646.63243	9.0657	1832.88431	9.02656	2163.99026	8.83868	3034.68728	8.85649
2	1864.50601	8.87347	2153.48053	8.88182	3071.17844	8.75126	1640.11669	9.10172
3	2181.72407	8.76684	3047.961	8.81792	1624.66693	9.18827	1851.9311	8.93372
4	3081.46691	8.72204	1625.83041	9.1817	1821.42329	9.08335	2163.05923	8.84249
5	1614.0182	9.24889	1839.90932	8.99209	2145.27297	8.9158	3057.11198	8.79152
Average	**2077.66952**	**8.93539**	**2100.01311**	**8.98002**	**2165.30638**	**8.95547**	**2349.38126**	**8.90519**

4.1.3 Observations

- The aggregate throughput and throughput is almost same for both the case.
- However, throughput is slightly more for the case HIJK. The throughput of HIJK is higher by 0.3% than HKMN.
- The reason for the similarity is due to the network topology and all the links have the corresponding links in HIJK and HKMN have the same bandwidth, delay and losses.
- However, the increase in throughput in the case of HIJK might be due to the same grand parental switch (S2) while it is different (S1) in the case of HKMN.

4.2 (L) Custom Topology ii) Horizontal Scaling and Load Balancing




```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

   net = Mininet( topo=None,
                  build=False,
                  ipBase='10.0.0.0/8')

   info( '*** Adding controller\n' )
   c0=net.addController(name='c0',
                        controller=Controller,
                        protocol='tcp',
                        port=6633)

   info( '*** Add switches\n')
   s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
```

```
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    O = net.addHost('O', cls=Host, ip='10.0.0.9', defaultRoute=None)
    L = net.addHost('L', cls=Host, ip='10.0.0.6', defaultRoute=None)
    K = net.addHost('K', cls=Host, ip='10.0.0.5', defaultRoute=None)
    M = net.addHost('M', cls=Host, ip='10.0.0.7', defaultRoute=None)
    server3 = net.addHost('server3', cls=Host, ip='10.0.0.11', defaultRoute=None)
    N = net.addHost('N', cls=Host, ip='10.0.0.8', defaultRoute=None)
    server1 = net.addHost('server1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    J = net.addHost('J', cls=Host, ip='10.0.0.4', defaultRoute=None)
    server2 = net.addHost('server2', cls=Host, ip='10.0.0.10', defaultRoute=None)
    I = net.addHost('I', cls=Host, ip='10.0.0.3', defaultRoute=None)
    H = net.addHost('H', cls=Host, ip='10.0.0.2', defaultRoute=None)

    info( '*** Add links\n')
    s1s2 = {'bw':40}
    net.addLink(s1, s2, cls=TCLink , **s1s2)
    s2s4 = {'bw':20}
    net.addLink(s2, s4, cls=TCLink , **s2s4)
    s2s5 = {'bw':20}
    net.addLink(s2, s5, cls=TCLink , **s2s5)
    s4H = {'bw':10}
    net.addLink(s4, H, cls=TCLink , **s4H)
    s4I = {'bw':10}
    net.addLink(s4, I, cls=TCLink , **s4I)
    s5J = {'bw':10}
    net.addLink(s5, J, cls=TCLink , **s5J)
    s5K = {'bw':10}
    net.addLink(s5, K, cls=TCLink , **s5K)
    s6L = {'bw':10}
    net.addLink(s6, L, cls=TCLink , **s6L)
    s6M = {'bw':10}
    net.addLink(s6, M, cls=TCLink , **s6M)
    s7N = {'bw':10}
    net.addLink(s7, N, cls=TCLink , **s7N)
    s7O = {'bw':10}
    net.addLink(s7, O, cls=TCLink , **s7O)
    s1server1 = {'bw':40}
    net.addLink(s1, server1, cls=TCLink , **s1server1)
    s1s3 = {'bw':40}
    net.addLink(s1, s3, cls=TCLink , **s1s3)
    s3s6 = {'bw':20}
    net.addLink(s3, s6, cls=TCLink , **s3s6)
    s3s7 = {'bw':20}
    net.addLink(s3, s7, cls=TCLink , **s3s7)
    server2s6 = {'bw':40}
    net.addLink(server2, s6, cls=TCLink , **server2s6)
```

```
    s7server3 = {'bw':40}
    net.addLink(s7, server3, cls=TCLink , **s7server3)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s4').start([c0])
    net.get('s7').start([c0])
    net.get('s6').start([c0])
    net.get('s2').start([c0])
    net.get('s1').start([c0])
    net.get('s5').start([c0])
    net.get('s3').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

IP Address of Servers:
Server1 = 10.0.0.1
Server2 = 10.0.0.10
Server3 = 10.0.0.11

We will have to make the servers and the respective clients listen to these ports as the question demands.

4.2.1 Clients on HIJK

grace_c.sh

```
server1 python3.8 tcp_server_sc_thread1.py -p 0 &
server2 python3.8 tcp_server_sc_thread2.py -p 0 &
server3 python3.8 tcp_server_sc_thread3.py -p 0 &
H python3.8 tcp_client1.py -p 0 -i "1" &
I python3.8 tcp_client1.py -p 0 -i "2" &
J python3.8 tcp_client2.py -p 0 -i "3" &
K python3.8 tcp_client3.py -p 0 -i "4"
```

H,I -> Server1

J -> Server2

K -> Server3

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput
1	8780.50662	35.5012
2	8801.0025	35.33226
3	8779.57058	35.38937
4	8707.93701	35.61915
5	8800.12846	35.31352
Average	**8773.829034**	**35.4311**

Exp. No.	H		I		J		K	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	1626.79911	9.17623	1856.3807	8.91231	2166.73613	8.82748	3130.59068	8.58518
2	1841.46523	8.98449	2202.46029	8.6843	3094.26975	8.68595	1662.80723	8.97752
3	2150.46978	8.89425	3097.82958	8.67597	1688.19547	8.84251	1843.07575	8.97664
4	3018.73517	8.90329	1632.50113	9.14418	1910.78115	8.65857	2145.91956	8.91311
5	1658.76603	8.99939	1844.77854	8.96836	2225.65126	8.59381	3070.93263	8.75196
Average	**2059.24706**	**8.99153**	**2126.79005**	**8.87702**	**2217.12675**	**8.72166**	**2370.66517**	**8.84088**

4.2.2 Clients on HKMN

*grace_d.sh

```
server1 python3.8 tcp_server_sc_thread1.py -p 0 &
server2 python3.8 tcp_server_sc_thread2.py -p 0 &
server3 python3.8 tcp_server_sc_thread3.py -p 0 &
H python3.8 tcp_client1.py -p 0 -i "1" &
K python3.8 tcp_client1.py -p 0 -i "2" &
M python3.8 tcp_client2.py -p 0 -i "3" &
N python3.8 tcp_client3.py -p 0 -i "4"
```

H,K -> Server1

M -> Server2
N-> Server3

All the times are in ms, and all the throughputs are in Mb/s.

Exp. No.	Total Download Time	Aggregate throughput
1	8667.51838	35.88116
2	8748.63505	35.66369
3	8719.39825	35.71235
4	8670.32218	35.90369
5	8686.39994	35.82477
Average	**8698.45476**	**35.797132**

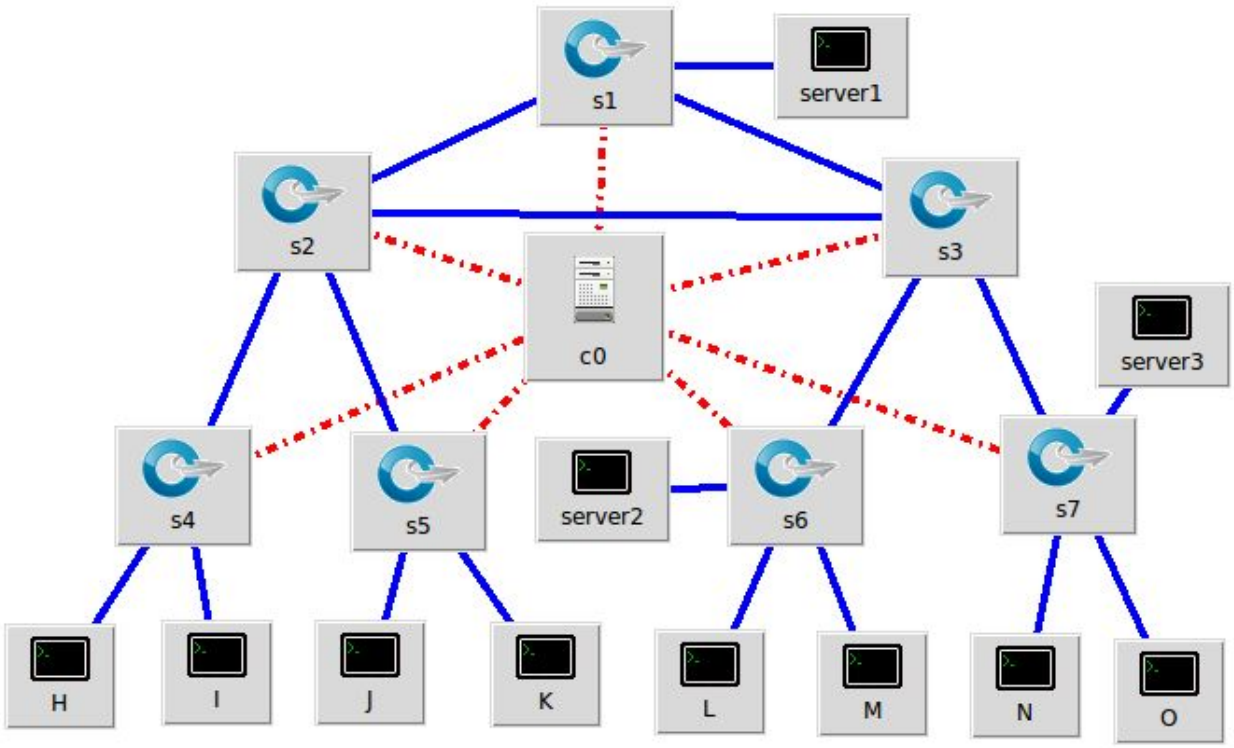
Exp. No.	H		K		M		N	
	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput	Dwld Time	Throughput
1	1622.31708	9.20158	1831.02369	9.03573	2169.34109	8.81688	3044.83652	8.82697
2	1846.81726	8.95846	2151.5491	8.88979	3135.20288	8.57255	1615.06581	9.24289
3	2163.03992	8.84257	3091.21513	8.69453	1627.78592	9.17067	1837.35728	9.00458
4	3066.08844	8.76578	1618.20745	9.22495	1828.9814	9.04582	2157.04489	8.86714
5	1603.81365	9.30774	1857.78904	8.90555	2170.76898	8.81108	3054.02827	8.8004
Average	**2060.41527**	**9.01523**	**2109.95688**	**8.95011**	**2186.41605**	**8.8834**	**2341.66655**	**8.9484**

4.2.3 Observation

- The aggregate throughput when run on hosts HKMN is higher than that of HIJK.
- It may be due to effect of addition of two other servers (server 1 and server 2) in the network and both being closer to M and N.

- From the previous experiments, we had shown that as links increase, throughput decreases. The same principle applies here.

4.3 (M) Custom Topology iii) With Loops



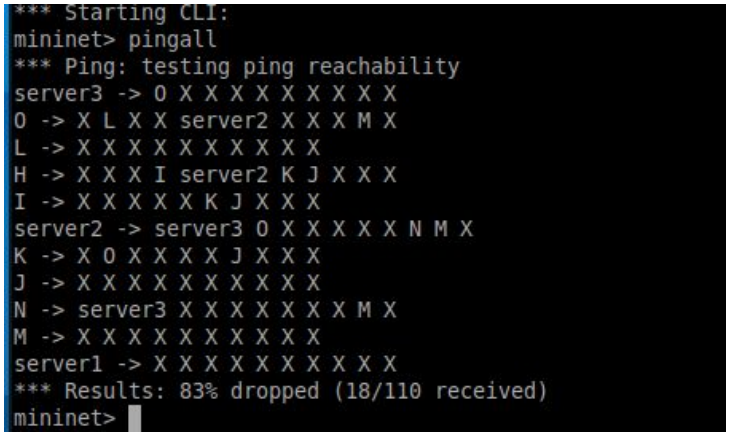

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

   net = Mininet( topo=None,
                  build=False,
                  ipBase='10.0.0.0/8')

   info( '*** Adding controller\n' )
   c0=net.addController(name='c0',
                        controller=Controller,
                        protocol='tcp',
                        port=6633)
```

```
    info( '*** Add switches\n')
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    H = net.addHost('H', cls=Host, ip='10.0.0.2', defaultRoute=None)
    server1 = net.addHost('server1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    L = net.addHost('L', cls=Host, ip='10.0.0.6', defaultRoute=None)
    O = net.addHost('O', cls=Host, ip='10.0.0.9', defaultRoute=None)
    N = net.addHost('N', cls=Host, ip='10.0.0.8', defaultRoute=None)
    K = net.addHost('K', cls=Host, ip='10.0.0.5', defaultRoute=None)
    M = net.addHost('M', cls=Host, ip='10.0.0.7', defaultRoute=None)
    I = net.addHost('I', cls=Host, ip='10.0.0.3', defaultRoute=None)
    server3 = net.addHost('server3', cls=Host, ip='10.0.0.11', defaultRoute=None)
    server2 = net.addHost('server2', cls=Host, ip='10.0.0.10', defaultRoute=None)
    J = net.addHost('J', cls=Host, ip='10.0.0.4', defaultRoute=None)

    info( '*** Add links\n')
    s1s2 = {'bw':40}
    net.addLink(s1, s2, cls=TCLink , **s1s2)
    s2s4 = {'bw':20}
    net.addLink(s2, s4, cls=TCLink , **s2s4)
    s2s5 = {'bw':20}
    net.addLink(s2, s5, cls=TCLink , **s2s5)
    s4H = {'bw':10}
    net.addLink(s4, H, cls=TCLink , **s4H)
    s4I = {'bw':10}
    net.addLink(s4, I, cls=TCLink , **s4I)
    s5J = {'bw':10}
    net.addLink(s5, J, cls=TCLink , **s5J)
    s5K = {'bw':10}
    net.addLink(s5, K, cls=TCLink , **s5K)
    s6L = {'bw':10}
    net.addLink(s6, L, cls=TCLink , **s6L)
    s6M = {'bw':10}
    net.addLink(s6, M, cls=TCLink , **s6M)
    s7N = {'bw':10}
    net.addLink(s7, N, cls=TCLink , **s7N)
    s7O = {'bw':10}
    net.addLink(s7, O, cls=TCLink , **s7O)
    s1server1 = {'bw':40}
    net.addLink(s1, server1, cls=TCLink , **s1server1)
    s1s3 = {'bw':40}
    net.addLink(s1, s3, cls=TCLink , **s1s3)
    s3s6 = {'bw':20}
    net.addLink(s3, s6, cls=TCLink , **s3s6)
    s3s7 = {'bw':20}
    net.addLink(s3, s7, cls=TCLink , **s3s7)
```

```
    server2s6 = {'bw':40}
    net.addLink(server2, s6, cls=TCLink , **server2s6)
    s7server3 = {'bw':40}
    net.addLink(s7, server3, cls=TCLink , **s7server3)
    s2s3 = {'bw':40}
    net.addLink(s2, s3, cls=TCLink , **s2s3)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s6').start([c0])
    net.get('s2').start([c0])
    net.get('s7').start([c0])
    net.get('s4').start([c0])
    net.get('s5').start([c0])
    net.get('s1').start([c0])
    net.get('s3').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

4.3.1 Observation

- The pings are not going between clients and the server. It is due to the loop present (s1s2s3) in the network. And because of it, the packets are not able to reach its destination The ping result can be seen below.[3]



[3] "Routing loop problem - Wikiwand." http://www.wikiwand.com/en/Routing_loop_problem. Accessed 25 Nov. 2020.