CSN-503

Large multimedia File Transfer

(Project no - 2)

Advanced Computer Networks

Group-8

- Bhanu Pratap - 17114019
- Sasi Kiran Bhimavarapu - 17114021
- Goddu Vishal - 17114034
- Jaynil Jaiswal - 17114040
- Ommi Akash - 17114053
- Shashank Kashyap - 17114070
- Vishnuteja Gattukindi - 17114031

Introduction

The objective of this project is to develop an application that transfers large multimedia files using sockets, and to perform a quantitative analysis on the impact of multithreading in file transfer using said application.

Contributions

Bhanu Pratap : Implemented simple FTP (Page 3-4)

Sasi Kiran : Handled file IO in FTP (Page 9-10, 15)

Goddu Vishal : Implemented multithreading FTP (Page 7-9)

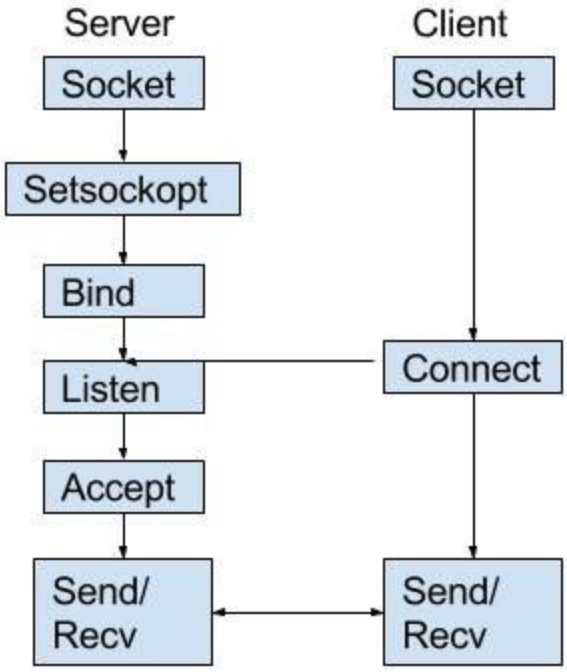
Jaynil Jaiswal : Analysis and benchmarking of application (Page 16-17)

Ommi Akash : Documentation and literature review (Page 5, 11)

Shashank Kashyap : Implemented multithreading FTP (Page 12-14)

Vishnuteja Gattukindi : Documentation and literature review (Page 5-6)

Socket programming



Socket programming is basically connecting two nodes in a network to enable communication with each other, a typical application has a socket which listens, the server, on a particular IP, while another socket, the client, reaches out to form a connection.

We begin by making a basic socket

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Here AF_INET refers to ipv4 and SOCK_STREAM tells us that it is a connection oriented TCP protocol.

Once we make our socket we must bind our socket to an IP and a port this is done by

```
server.bind((LOCALHOST, PORT))
```

```
server.listen(5)
```

Here LOCALHOST is the IP and PORT is the port we are transferring data through

LOCALHOST is set to 127.0.0.1 (loopback address), since both server and client are being run on the same system, so our server is listening on IP 127.0.0.1 and port 8080.

Once our server is active, any client can request to connect to our server using this set of IP and port number.

Before we try to connect to a server we must also create a socket for the client as we have done for the server. And this time instead of binding it to a specific (IP, port) pair at the client side, we request a connection (using any unused port (randomly) at client side) to the specific Server.

```
client.connect((SERVER, PORT))
```

This line of code when run from the server application requests a connection to our server.

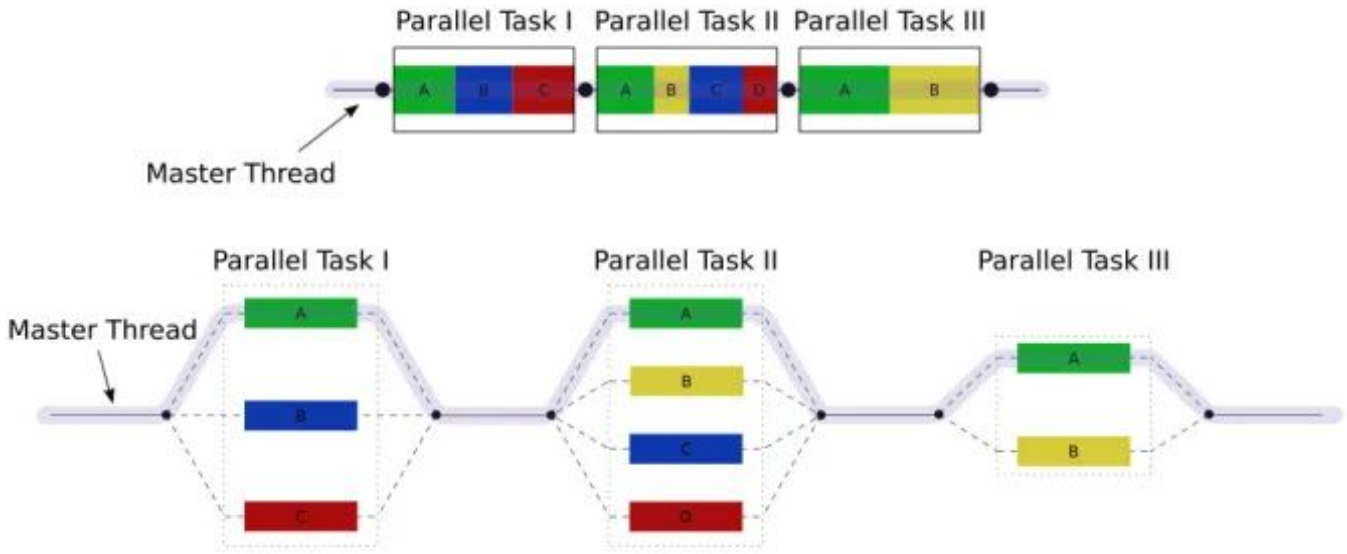
Once our client requests a connection our server can accept this request using the server.accept() method.

```
clientsock, clientAddress = server.accept()
```

When the server accepts a connection request, it creates a new socket object for communication with the client.

Once a connection has been established between our client and server the client can send data using ***sendall()*** method and the server receives this using ***recv()***, this will be looked at in more detail once we look at the code of our application.

Threads



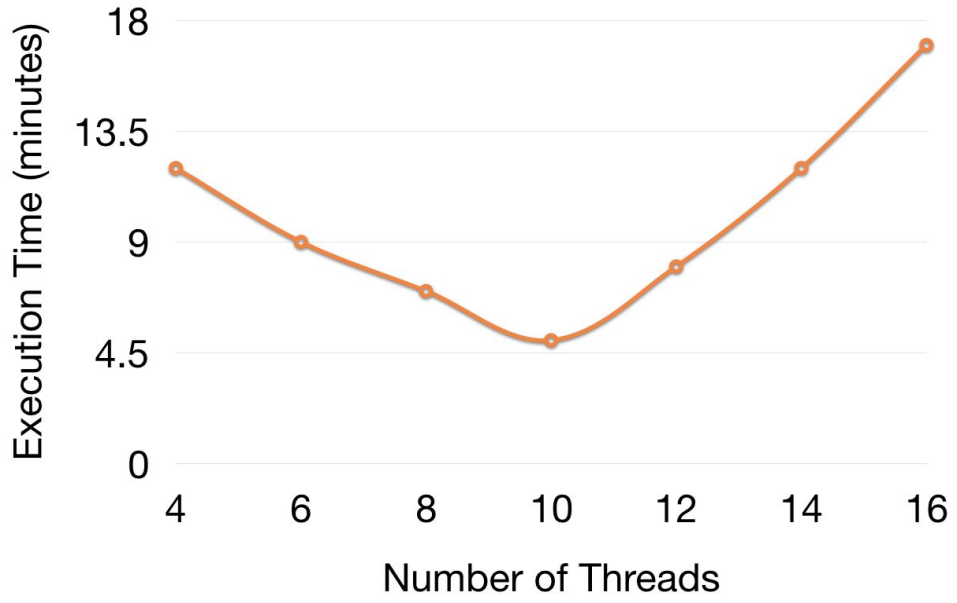
Multithreading is basically the process of splitting a task into multiple parallel tasks as shown in the figure. Each parallel task is run on its own thread, hence the name multithreading. The advantage of multithreading is fragmenting our primary task into smaller subtasks that can be run parallely, as opposed to the sequential nature of a single thread. When this happens instead of processing a total of 12 seconds that task 1 takes in a single thread, it can be split into three parallel threads each with a computation time of 4s. Which is a considerable decrease in the processing time.

The same concept can be used in media transfer, by fragmenting and sending chunks of data simultaneously in parallel, as opposed to sequentially sending our entire data at once. Each thread transfers a chunk of data from the client to the server and once all the data has been received by the server it merges these chunks and puts together our received media.

Since we have buffers in our sending and receiving methods of socket programming the chunk sizes must be given careful consideration which will be discussed further along this report.

Effect of no of threads:

The first thought that might occur when reading about threading is, why not use as many threads as possible, shouldn't the parallelization reduce our time considerably?



But the graph shown above indicates that this clearly is not the case. The reason this happens is when we split our process/data an overhead of allocating threads to the fragmented subtasks and merging them. So if we keep increasing the number of threads the overhead that is produced could deem to be counterproductive in reducing the processing time, which is essentially us forgetting the brief.

Threads in Python

Python has a 'Thread' class which represents an activity that is run in a separate thread of control. To specify the activity done by the thread, we override the ***__init__()*** and ***run()*** methods of this class.

__init__() is invoked when a thread object is created, this thread's activity is started by calling ***start()*** method of Thread class. This invokes the ***run()*** method in a separate thread of control. The thread remains 'alive' until its ***run()*** method terminates - either normally, or by raising an unhandled exception.

Another method used in our application is the ***join()*** method, in which other threads can call a thread's ***join()*** method. This blocks the calling thread until the thread whose ***join()*** method is called is terminated.

Code

With the basics of threads and socket programming covered we can move to the application that we have written.

Client.py

```
import socket
import threading
import time
from struct import pack

SERVER = "127.0.0.1"
PORT = 8080

class ServerThread(threading.Thread):
    def __init__(self, data_id, message, file_type):
        threading.Thread.__init__(self)
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((SERVER, PORT))
        self.message = message
        self.data_id = data_id
```

```
        self.file_type = file_type
        print("Created a new client and connected to server")

    def run(self):
        bdata_id = self.data_id.to_bytes(16, 'big')
        self.client.sendall(bdata_id)

        bfile_type = self.file_type.to_bytes(16, 'big')
        self.client.sendall(bfile_type)

        length = len(self.message).to_bytes(16, 'big')
        print(length)
        print(f"Chunk of size {int.from_bytes(length, 'big')} bytes to be
sent")
        self.client.sendall(length)

        self.client.sendall(self.message)
        print(f"Chunk sent")

        while True:
            in_data =  self.client.recv(1024)
            print("From Server :" ,in_data.decode())
            if in_data.decode('utf-8')=='transmission done':
                break
        self.client.close()

file_to_send = 'big.jpg'
file_type = 1 if file_to_send.split('.')[1]=='mp4' else 0
with open(file_to_send, "rb") as media:
  f = media.read()

total_data = bytearray(f)
total_file_size = len(total_data)
print(f"Total filesize is {total_file_size}")

NUM_THREADS = 8
THREADS = []
```

```
start = time.time()

for thr_id in range(NUM_THREADS):
    if thr_id==NUM_THREADS-1:
        thr_data =
total_data[thr_id*int(total_file_size/NUM_THREADS):total_file_size]
    else:
        thr_data =
total_data[thr_id*int(total_file_size/NUM_THREADS):(thr_id+1)*int(total_fi
le_size/NUM_THREADS)]
    thread = ServerThread(message=thr_data, data_id=thr_id+1,
file_type=file_type)
    THREADS.append(thread)
    thread.start()

for thr_id in THREADS:
    thr_id.join()

end = time.time()

print(f"File transfer done in {end-start} seconds")
```

To begin with we set up a socket for the client as discussed in the socket programming section of the report. Since we modify the threads we call the method

```
threading.Thread.__init__(self)
```

Here self.message and self.data_id are instance variables of our client thread.

Since the buffer size of our client and server is 16 bytes we try to maintain a standard of 16 bytes for all information that is passed through this connection. Hence we convert the length of our message and the data_id into a byte stream of 16 bytes before we pass on this information to our server. This is done by overriding the ***run()*** method as discussed above in the threads section of the report.

```
bdata_id = self.data_id.to_bytes(16, 'big')
```

```
length = len(self.message).to_bytes(16, 'big')
```

The run function must also handle the response that our server sends when we have completed transferring the data

```
while True:
            in_data =  self.client.recv(1024)
            print("From Server :" ,in_data.decode())
            if in_data.decode('utf-8')=='transmission done':
                break
        self.client.close()
```

Once the __init___ and run() methods have been overridden we can start working on handling the file that has to be sent. And can also begin the timer for start of data transfer

```
file_to_send = 'big.jpg'
file_type = 1 if file_to_send.split('.')[1]=='mp4' else 0
with open(file_to_send, "rb") as media:
  f = media.read()
total_data = bytearray(f)
total_file_size = len(total_data)
start = time.time()
```

We begin by reading the file we send as a byte stream and once we have read the data we store it in the data variable as a bytearray.

Splitting into threads

```
for thr_id in range(NUM_THREADS):
    if thr_id==NUM_THREADS-1:
        thr_data =
total_data[thr_id*int(total_file_size/NUM_THREADS):total_file_size]
    else:
        thr_data =
total_data[thr_id*int(total_file_size/NUM_THREADS):(thr_id+1)*int(total_fi
le_size/NUM_THREADS)]
    thread = ServerThread(message=thr_data, data_id=thr_id+1)
    THREADS.append(thread)
    thread.start()
```

Once our data is stored in a byte stream the next step is to split the data into chunks so that our threads can share each chunk, what the above code does is split our data into chunks, but datalen%num_threads doesn't always have to be 0 so what this does is if we have data of 20.3 mb being split into 4 threads, it makes the first 3 threads pass chunks of 5 mb and the last thread pass a chunk of 5.3 mb. Once this distribution of data is done we append each thread to THREADS and call the ***start()*** method of our thread class, this inturn calls the run() method that we overrode to make our data transfer.

```
for thr_id in THREADS:
    thr_id.join()
```

Once all the chunks have been transferred we join the threads, this makes sure that all of our chunks have been transferred(explained in threads section), and we can check the end time of our transfer.

```
end = time.time()
```

Server.py

```
import socket
import threading
from struct import unpack
import io
import time
import PIL.Image as Image

NUM_THREADS = 8
def merge_img():
    global received
    n = 0
    while True:
        if len(received.file_pieces)==NUM_THREADS:
            n += 1
            final_file = b''
            for key in sorted(received.file_pieces.keys()):
                v = received.file_pieces[key]
                print(f"key = {key} data = {len(v)}")
```

```
                final_file += v
            print(len(final_file))

            if received.file_type==0:
                image = Image.open(io.BytesIO(final_file))
                image.save(f'received_{n}.jpg')
            else:
                with open(f'received_{n}.mp4', 'wb') as video:
                    video.write(final_file)
            received.file_pieces = {}

class ClientThread(threading.Thread):
    def __init__(self,clientAddress,clientsocket):
        threading.Thread.__init__(self)
        self.csocket = clientsocket
        print ("New connection added: ", clientAddress)
    def run(self):
        print ("Connection from : ", clientAddress)
        #self.csocket.send(bytes("Hi, This is from Server..",'utf-8'))
        msg = ''

        bdata_id = self.csocket.recv(16)
        data_id = int.from_bytes(bdata_id, 'big')

        bfile_type = self.csocket.recv(16)
        file_type = int.from_bytes(bfile_type, 'big')

        chunk_size = self.csocket.recv(16)
        length = int.from_bytes(chunk_size, 'big')
        print(chunk_size)
        print(f"Chunk of size {length} bytes to be recieved")

        data = b''
        while len(data) < length:
            # doing it in batches is generally better than trying
            # to do it all in one go, so I believe.
            to_read = length - len(data)
```

```
            data += self.csocket.recv(4096 if to_read > 4096 else to_read)

        # send our "done" ack
        print(f"Chunk number {data_id} of size {len(data)} bytes
recieved")
        received.file_pieces[data_id] = data
        received.file_type = file_type

        msg = "transmission done"
        self.csocket.send(msg.encode('utf-8'))


        print ("Client at ", clientAddress , " disconnected...")

LOCALHOST = "127.0.0.1"
PORT = 8080
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((LOCALHOST, PORT))
print("Server started")
print("Waiting for client request..")

class Received:
    def __init__(self):
        self.file_pieces = {}
        self.file_type = None
received = Received()
t1 = threading.Thread(target=merge_img, args=[])
t1.start()

while True:
    server.listen(1)
    clientsock, clientAddress = server.accept()
    newthread = ClientThread(clientAddress, clientsock)
    newthread.start()
```

Similar to the client we begin with setting up a socket for our server.

Similarly we override the __init__ and run methods.

```
def run(self):
        print ("Connection from : ", clientAddress)
              msg = ''
        while True:
            bdata_id = self.csocket.recv(16)
            data_id = int.from_bytes(bdata_id, 'big')
            chunk_size = self.csocket.recv(16)
            print(chunk_size)
            length = int.from_bytes(chunk_size, 'big')
            print(f"Chunk of size {length} bytes to be recieved")
            data = b''
            while len(data) < length:
                to_read = length - len(data)
                data += self.csocket.recv(4096 if to_read > 4096 else
to_read)
            print(f"Chunk number {data_id} of size {len(data)} bytes
recieved")
            recieved_file_pieces[data_id] = data
            msg = "transmission done"
            self.csocket.send(msg.encode('utf-8'))
        print ("Client at ", clientAddress , " disconnected...")
```

What this is doing is receiving our chunks and data_id as a bytestream of 16 bytes and the data of this file_piece is being stored in the key of data_id, as seen in the code we receive the data in batches of 4096 as it is a good practice for performance. And if we receive the total length we send a “transmission done” message to our client.

Once we have received all the chunks of our large file on the server, we also need to assemble them in the correct order to obtain back the original file. As the server is always listening for new connections on the master thread, we can’t check for file transfer completion there. Thus, we create a seperate thread t1 which targets a function that handles this merging task.

```
t1 = threading.Thread(target=merge_img, args=[recieved_file_pieces])
```

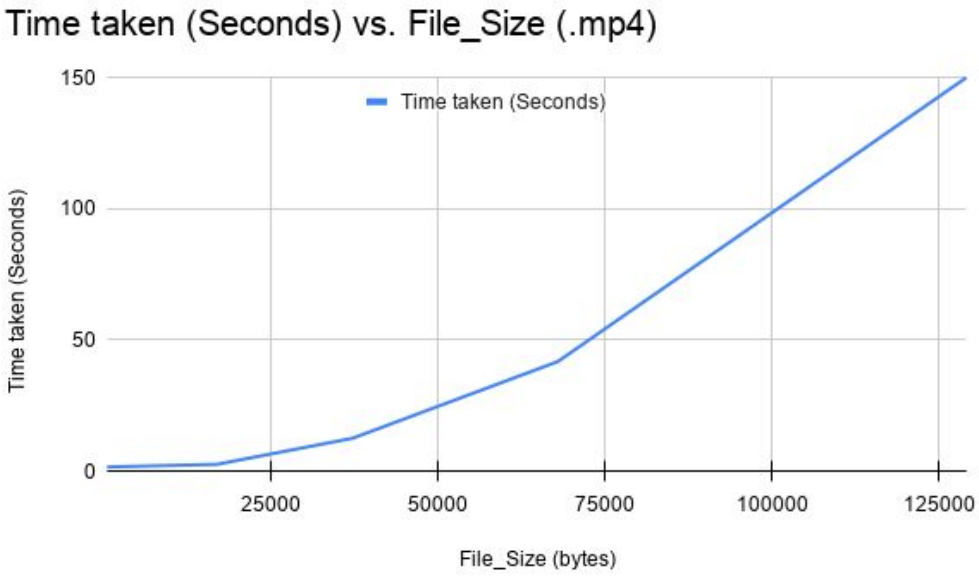
```
t1.start()
```

Once t1.start() it continuously checks the ‘recieved_file_pieces’ buffer (that is intended to store the original file) for the arrival of its children chunks. Once recieved_file_pieces has all the chunks (we assume that the server knows the number of chunks the client will be using for the file transfer), we merge them in the correct order into a single buffer ‘final file’ (consisting of bytes) and save it into storage with the correct file extension.

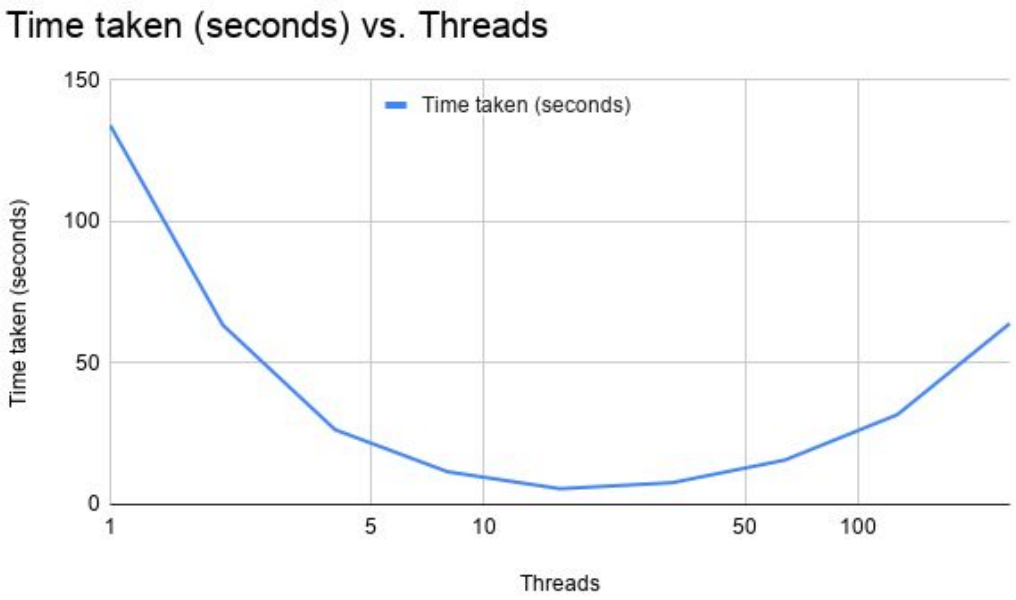
```
def merge_img():
    global received
    n = 0
    while True:
        if len(received.file_pieces)==NUM_THREADS:
            n += 1
            final_file = b''
            for key in sorted(received.file_pieces.keys()):
                v = received.file_pieces[key]
                print(f"key = {key} data = {len(v)}")
                final_file += v
            print(len(final_file))

            if received.file_type==0:
                image = Image.open(io.BytesIO(final_file))
                image.save(f'received_{n}.jpg')
            else:
                with open(f'received_{n}.mp4', 'wb') as video:
                    video.write(final_file)
            received.file_pieces = {}
```

Upon completion of merging, we erase the ‘recieved_file_pieces’ buffer content, so that it can be used for another file transfer and continue the thread.

Analysis

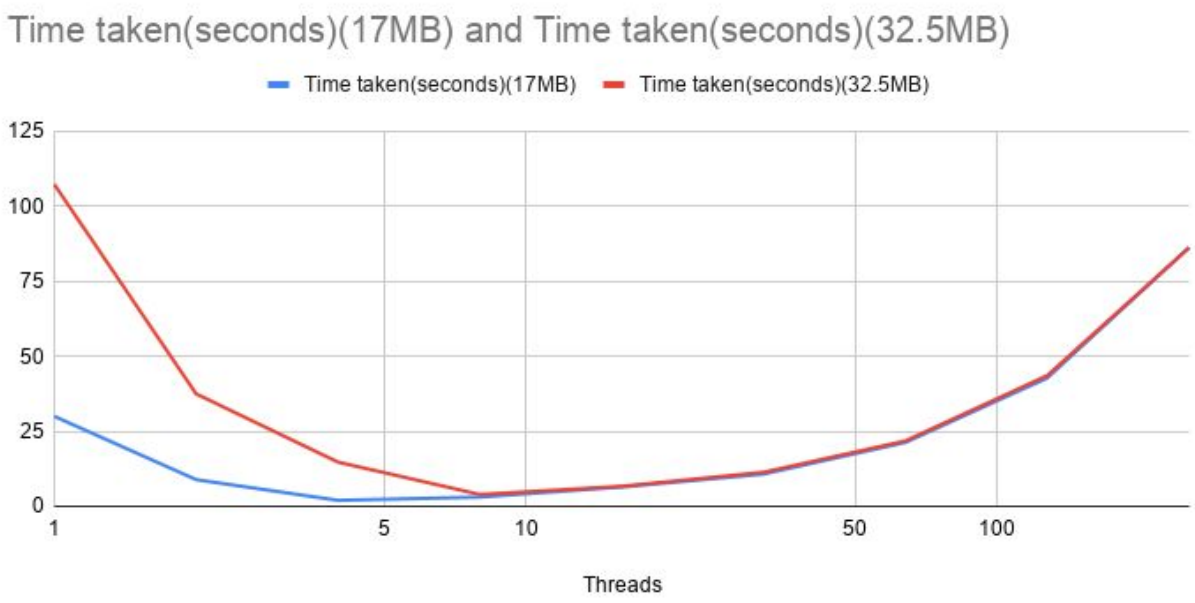


As expected when the file size increases, with a single thread, the time taken for the file to transfer increases. The reasoning behind this is pretty straight forward, as more data needs to be sent hence more time is required.



Like we discussed in the threading section of our report, as the number of threads increase we see a decrease in the time taken for the file to transfer but a clear dip is visible in this graph, the rise occurs due to the overhead that is produced by introducing an excessive amount of threads to share a file that probably doesn't require as many threads as we create. Hence the usage of the number of threads can be optimised based on the file size that we ought to transfer.

Time taken for different file sizes



When a large number of threads are used, the time taken by each thread to finish transmitting a chunk becomes negligible, compared to the overhead due to creation of a large number of threads. Therefore, file size becomes immaterial and both times converge as we increase the number of threads beyond a limit.