Computer Networks Assignment 3

Navneel Singhal

November 21, 2020

Contents

1 Code

```
import socket
import hashlib
import collections
import threading
import sys
import csv
import time
import matplotlib.pyplot as plt

'''
HTTP/1.1 206 Partial Content
Date: Wed, 18 Nov 2020 06:30:10 GMT
Server: Apache/2.4.29 (Ubuntu)
Last-Modified: Mon, 22 Apr 2019 17:44:41 GMT
ETag: "63025a-5872205e3b440"
Accept-Ranges: bytes
Content-Length: 100
Vary: Accept-Encoding
Content-Range: bytes 0-99/6488666
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/plain

'''

def parseCompletely(clientSocket):
    messageSize = int(1e9)
    headerSoFar = bytearray(b'')
    messageSoFar = bytearray(b'')
    state = 0 # 0 if inside header and 1 if in body
    while len(messageSoFar) < messageSize:
        receivedMessage = clientSocket.recv(1024)
        if receivedMessage == b'':
            raise RuntimeError('Broken connection')
        if state == 0:
            headerSoFar += bytearray(receivedMessage)
```

```
            pos = headerSoFar.find(b'\r\n\r\n')
            if pos != -1:
                messageSoFar = headerSoFar[pos + 4:]
                headerSoFar = headerSoFar[:pos + 4].decode('ASCII')
                lengthKeyPosition = headerSoFar.find('Content-Length: ')
                assert lengthKeyPosition != -1
                startValuePosition = lengthKeyPosition + len('Content-Length: ')
                endValuePosition = startValuePosition
                while headerSoFar[endValuePosition] != '\r':
                    endValuePosition += 1
                messageSize = int(headerSoFar[startValuePosition : endValuePosition])
                state = 1
                headerSoFar = bytearray(b'')
        else:
            messageSoFar += bytearray(receivedMessage)
    return messageSoFar

def downloadAtOnce():
    serverName = 'vayu.iitd.ac.in'
    serverPort = 80
    message = "GET /big.txt HTTP/1.1\r\nHost: vayu.iitd.ac.in\r\nConnection: keep-alive\r\n\r\n".encode('ASCII')
    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clientSocket.connect((serverName, serverPort))
    clientSocket.send(message)
    receivedMessage = parseCompletely(clientSocket)
    clientSocket.close()
    hashValue = hashlib.md5(bytes(receivedMessage)).hexdigest()
    assert hashValue == '70a4b9f4707d258f559f91615297a3ec'

def downloadChunkWise():
    serverName = 'vayu.iitd.ac.in'
    serverPort = 80
    message = "GET /big.txt HTTP/1.1\r\nHost: vayu.iitd.ac.in\r\nConnection: keep-alive\r\nRange: bytes="
    messageEnd = "\r\n\r\n"
    chunkSize = 1024 * 16
    fileSize = 6488666
    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clientSocket.connect((serverName, serverPort))
    receivedMessage = bytearray(b'')
    missed = 0
    for currentPtr in range(0, fileSize, chunkSize):
        sendMessage = message + str(currentPtr) + '-' + str(min(currentPtr + chunkSize, fileSize) - 1) + messageEnd
        clientSocket.send((sendMessage).encode('ASCII'))
        while True:
            try:
                x = parseCompletely(clientSocket)
                receivedMessage += x
                break
            except:
                missed += 1
                clientSocket.close()
                clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                clientSocket.connect((serverName, serverPort))
                sendMessage = message + str(currentPtr) + '-' + str(min(currentPtr + chunkSize, fileSize) - 1) + messageEnd
                clientSocket.send((sendMessage).encode('ASCII'))
    clientSocket.close()
    hashValue = hashlib.md5(bytes(receivedMessage)).hexdigest()
    assert hashValue == '70a4b9f4707d258f559f91615297a3ec'

def downloadPipelined():
    serverName = 'vayu.iitd.ac.in'
    serverPort = 80
    message = "GET /big.txt HTTP/1.1\r\nHost: vayu.iitd.ac.in\r\nConnection: keep-alive\r\nRange: bytes="
    messageEnd = "\r\n\r\n"
    fileSize = 6488666
    chunkSize = 1024 * 16 # 16 KB
    pipelineSize = 10
    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clientSocket.connect((serverName, serverPort))
    receivedMessage = bytearray(b'')
    chunkSet = collections.OrderedDict()
    totalChunks = 0
    for currentPtr in range(0, fileSize, chunkSize):
        chunkSet[(currentPtr, min(currentPtr + chunkSize, fileSize) - 1)] = None
        totalChunks += 1
    d = dict() # mapping from chunk ranges to chunks
```

```
    currentChunkRange = None # a pair of integers determining the chunk
    messageSoFar = bytearray(b'')
    headerSoFar = bytearray(b'')
    state = 0 # 0 for inside header and 1 for inside message
    messageSize = int(1e9) # total message size - whenever the state is 0, change it to 1e9
    completeChunks = 0 # whenever we read a chunk completely, add it to the dictionary and increment this variable by 1
    currentlyParsing = False
    totalRightNow = 0
    received = min(totalChunks, pipelineSize)
    while completeChunks < totalChunks:
        while not currentlyParsing:
            try:
                if received == min(totalChunks - completeChunks, pipelineSize):
                    received = 0
                    sent = 0
                    for (begin, end) in chunkSet:
                        sendMessage = message + str(begin) + '-' + str(end) + messageEnd
                        clientSocket.send((sendMessage).encode('ASCII'))
                        sent += 1
                        if sent == pipelineSize:
                            break
                receivedMessage = bytearray(clientSocket.recv(1024))
                if receivedMessage == bytearray(b''):
                    raise RuntimeError('Broken connection')
                currentlyParsing = True
            except:
                currentChunkRange = None
                messageSoFar = bytearray(b'')
                headerSoFar = bytearray(b'')
                state = 0
                messageSize = int(1e9)
                clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                clientSocket.connect((serverName, serverPort))
                sent = 0
                for (begin, end) in chunkSet:
                    sendMessage = message + str(begin) + '-' + str(end) + messageEnd
                    clientSocket.send((sendMessage).encode('ASCII'))
                    sent += 1
                    if sent == pipelineSize:
                        break
        if state == 0:
            assert len(receivedMessage) != 0
            headerSoFar += receivedMessage
            pos = headerSoFar.find(b'\r\n\r\n')
            if pos != -1:
                receivedMessage = headerSoFar[pos + 4:]
                headerSoFar = headerSoFar[:pos + 4].decode('ASCII')
                lengthKeyPosition = headerSoFar.find('Content-Length: ')
                assert lengthKeyPosition != -1
                startValuePosition = lengthKeyPosition + len('Content-Length: ')
                endValuePosition = startValuePosition
                while headerSoFar[endValuePosition] != '\r':
                    endValuePosition += 1
                messageSize = int(headerSoFar[startValuePosition : endValuePosition])
                rangeKeyPosition = headerSoFar.find('Content-Range: bytes ')
                assert rangeKeyPosition != -1
                startValuePosition = rangeKeyPosition + len('Content-Range: bytes ')
                endValuePosition = startValuePosition
                while headerSoFar[endValuePosition] != '/':
                    endValuePosition += 1
                currentChunkRange = tuple([int(x) for x in headerSoFar[startValuePosition : endValuePosition].split('-')])
                state = 1
                headerSoFar = bytearray(b'')
                messageSoFar = bytearray(b'')
            else:
                receivedMessage = bytearray()
                currentlyParsing = False
                continue
        if state == 1:
            i = 0
            while len(messageSoFar) < messageSize and i < len(receivedMessage):
                messageSoFar += receivedMessage[i : i + 1]
                i += 1
            receivedMessage = receivedMessage[i:]
            if len(messageSoFar) == messageSize:
                d[currentChunkRange] = messageSoFar
```

```
                chunkSet.pop(currentChunkRange)
                currentChunkRange = None
                messageSoFar = bytearray(b'')
                headerSoFar = bytearray(b'')
                state = 0
                messageSize = int(1e9)
                completeChunks += 1
                received += 1
            if len(receivedMessage) == 0:
                currentlyParsing = False
                continue
    ans = bytearray(b'')
    for key in sorted(d):
        ans += d[key]
    hashValue = hashlib.md5(bytes(ans)).hexdigest()
    assert hashValue == '70a4b9f4707d258f559f91615297a3ec'

'''
variables to be synchronized - chunkSet, completeChunks, d (not necessary since d is an array)
'''

lockSet = threading.Lock()
lockComplete = threading.Lock()

chunkSet = collections.OrderedDict()
totalChunks = 0
completeChunks = 0
fileSize = -1 #6488666
chunkSize = 1024 * 16
d = []
timeStamps = [[] for i in range(10000)]
startTime = time.time()

def chunkAllocator(pipelineSize):
    global chunkSet
    localChunkSet = collections.OrderedDict()
    lockSet.acquire()
    for (begin, end) in chunkSet:
        localChunkSet[(begin, end)] = None
        if len(localChunkSet) == pipelineSize:
            break
    for (begin, end) in localChunkSet:
        chunkSet.pop((begin, end))
    lockSet.release()
    return localChunkSet

def downloadSingleThread(threadNumber, serverName, pipelineSize, fileName):
    global totalChunks, completeChunks, d, fileSize, chunkSize
    serverPort = 80
    message = "GET " + fileName + " HTTP/1.1\r\nHost: " + serverName + "\r\nConnection: keep-alive\r\nRange: bytes="
    messageEnd = "\r\n\r\n"
    chunkSize = 1024 * 16 # 16 KB
    while True:
        try:
            clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            clientSocket.settimeout(2)
            clientSocket.connect((serverName, serverPort))
            clientSocket.settimeout(None)
            break
        except:
            continue
    sizeProcessed = 0
    headerSize = 0
    receivedMessage = bytearray(b'')
    currentChunkRange = None # a pair of integers determining the chunk
    messageSoFar = bytearray(b'')
    headerSoFar = bytearray(b'')
    state = 0 # 0 for inside header and 1 for inside message
    messageSize = int(1e9) # total message size - whenever the state is 0, change it to 1e9
    currentlyParsing = False
    totalRightNow = 0
    localChunkSet = collections.OrderedDict()
    while True:
        #print(threadNumber)
        lockComplete.acquire()
        breakCondition = not (completeChunks < totalChunks)
```

```
        lockComplete.release()
        if breakCondition:
            return
        while not currentlyParsing:
            try:
                lockComplete.acquire()
                breakCondition = not (completeChunks < totalChunks)
                lockComplete.release()
                if breakCondition:
                    return
                if len(localChunkSet) == 0:
                    localChunkSet = chunkAllocator(pipelineSize)
                    length = len(localChunkSet)
                    if len(localChunkSet) == 0:
                        return
                    for (begin, end) in localChunkSet:
                        sendMessage = message + str(begin) + '-' + str(end) + messageEnd
                        clientSocket.send((sendMessage).encode('ASCII'))
                receivedMessage = bytearray(clientSocket.recv(1024))
                if receivedMessage == bytearray(b''):
                    raise RuntimeError('Broken connection')
                currentlyParsing = True
            except:
                try:
                    currentChunkRange = None
                    messageSoFar = bytearray(b'')
                    headerSoFar = bytearray(b'')
                    state = 0
                    messageSize = int(1e9)
                    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    clientSocket.settimeout(2)
                    clientSocket.connect((serverName, serverPort))
                    clientSocket.settimeout(None)
                    for (begin, end) in localChunkSet:
                        sendMessage = message + str(begin) + '-' + str(end) + messageEnd
                        clientSocket.send((sendMessage).encode('ASCII'))
                    continue
                except:
                    continue
        if state == 0:
            assert len(receivedMessage) != 0
            headerSoFar += receivedMessage
            pos = headerSoFar.find(b'\r\n\r\n')
            if pos != -1:
                receivedMessage = headerSoFar[pos + 4:]
                headerSoFar = headerSoFar[:pos + 4].decode('ASCII')
                lengthKeyPosition = headerSoFar.find('Content-Length: ')
                assert lengthKeyPosition != -1
                startValuePosition = lengthKeyPosition + len('Content-Length: ')
                endValuePosition = startValuePosition
                while headerSoFar[endValuePosition] != '\r':
                    endValuePosition += 1
                messageSize = int(headerSoFar[startValuePosition : endValuePosition])
                rangeKeyPosition = headerSoFar.find('Content-Range: bytes ')
                assert rangeKeyPosition != -1
                startValuePosition = rangeKeyPosition + len('Content-Range: bytes ')
                endValuePosition = startValuePosition
                while headerSoFar[endValuePosition] != '/':
                    endValuePosition += 1
                currentChunkRange = tuple([int(x) for x in headerSoFar[startValuePosition : endValuePosition].split('-')])
                state = 1
                headerSoFar = bytearray(b'')
                messageSoFar = bytearray(b'')
                headerSize = pos + 4
            else:
                receivedMessage = bytearray()
                currentlyParsing = False
                continue
        if state == 1:
            i = 0
            while len(messageSoFar) < messageSize and i < len(receivedMessage):
                messageSoFar += receivedMessage[i : i + 1]
                i += 1
            receivedMessage = receivedMessage[i:]
            if len(messageSoFar) == messageSize:
                d[currentChunkRange[0] // chunkSize] = messageSoFar
```

```
                localChunkSet.pop(currentChunkRange)
                sizeProcessed += messageSize + headerSize
                timeStamps[threadNumber].append((time.time() - startTime, sizeProcessed))
                currentChunkRange = None
                messageSoFar = bytearray(b'')
                headerSoFar = bytearray(b'')
                state = 0
                messageSize = int(1e9)
                lockComplete.acquire()
                completeChunks += 1
                #print('Completed chunks, Thread number:', completeChunks, threadNumber)
                lockComplete.release()
            if len(receivedMessage) == 0:
                currentlyParsing = False
                continue
    return

def getContentLength(clientSocket):
    global fileSize
    headerSoFar = ''
    while True:
        receivedMessage = clientSocket.recv(1024)
        if receivedMessage == b'':
            raise RuntimeError('Broken connection')
        headerSoFar += (receivedMessage).decode('ASCII')
        if headerSoFar.lower().find('404 not found') != -1:
            fileSize = -1
            return
        pos = headerSoFar.find('\r\n\r\n')
        if pos != -1:
            headerSoFar = headerSoFar[:pos + 4]
            lengthKeyPosition = headerSoFar.find('Content-Length: ')
            assert lengthKeyPosition != -1
            startValuePosition = lengthKeyPosition + len('Content-Length: ')
            endValuePosition = startValuePosition
            while headerSoFar[endValuePosition] != '\r':
                endValuePosition += 1
            fileSize = int(headerSoFar[startValuePosition : endValuePosition])
            return
    fileSize = -1
    return

def downloadMultipleConnections(l):
    global totalChunks, completeChunks, d, fileSize, chunkSize
    assert len(l) >= 1
    # find fileSize by sending request to the first server in l
    for ((serverName, filePath), _) in l:
        # we can test for whether the resource is at these servers by sending a head request
        # but we will assume that it exists, raising an exception if it is not at the first one
        serverPort = 80
        message = "HEAD " + filePath + " HTTP/1.1\r\nHost: " + serverName + "\r\nConnection: keep-alive\r\n\r\n"
        clientSocket = None
        while True:
            try:
                getContentLength(clientSocket)
                break
            except:
                try:
                    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    clientSocket.connect((serverName, serverPort))
                    clientSocket.send(message.encode('ASCII'))
                    continue
                except:
                    continue
        break
    if fileSize == -1:
        raise RuntimeError('File not found at the server', serverName)
    print('File size:', fileSize)
    d = [None for i in range(0, fileSize, chunkSize)]
    for currentPtr in range(0, fileSize, chunkSize):
        chunkSet[(currentPtr, min(currentPtr + chunkSize, fileSize) - 1)] = None
        totalChunks += 1
    pipelineSize = 1
    t = []
    threadCount = 0
    for ((url, filePath), numThreads) in l:
```

```
            t += [threading.Thread(target=downloadSingleThread, args=(threadCount + i, url, pipelineSize, filePath)) for i in range
            threadCount += numThreads
        for th in t:
            th.start()
        for th in t:
            th.join()
        ans = bytearray(b'')
        for k in d:
            ans += k
        hashValue = hashlib.md5(bytes(ans)).hexdigest()
        print('File hash:', hashValue)

    def parseUrl(url):
        # http:// or https://
        if len(url) >= 7 and url[:7] == 'http://':
            url = url[7:]
        elif len(url) >= 8 and url[:8] == 'https://':
            url = url[8:]
        pos = url.find('/')
        assert pos != -1
        return (url[:pos], url[pos:])

    def main():
        #downloadAtOnce()
        #downloadChunkWise()
        #downloadPipelined()
        #l = [(('vayu.iitd.ac.in', '/big.txt'), 10)]
        #l = [(('norvig.com', '/big.txt'), 10)]
        assert len(sys.argv) > 1
        with open(sys.argv[1], newline='') as csvfile:
            reader = csv.reader(csvfile, delimiter=',')
            l = [(parseUrl(line[0]), int(line[1])) for line in list(reader)]
            downloadMultipleConnections(l)
            plotData = [list(zip(*timeStamps[i])) for i in range(sum([line[1] for line in l]))]
            for i in range(l[0][1]):
                plt.plot(plotData[i][0], plotData[i][1], color='red')
            if len(l) > 1:
                for i in range(l[0][1], l[0][1] + l[1][1]):
                    plt.plot(plotData[i][0], plotData[i][1], color='blue')
            if len(sys.argv) > 2:
                plt.savefig(sys.argv[2])
            else:
                plt.show()

    if __name__ == '__main__':
        main()
```

2 Part 1

Problem: Use TCP sockets to download a file in one go, ensuring that the file is received correctly by verifying file hashes.

Solution: The solution can be seen as in the function `downloadAtOnce()` in the code. It is quite simple; it creates a socket, connects, sends a request, receives the response, parses the message and closes the connection, and verifies the file hash as well to ensure that the file is received correctly (we always do this check whenever we download a file).

3 Part 2

Problem: Use the HTTP header fields to download the file in chunks.

Solution: The solution for this can be seen as in the function `downloadChunkWise()` in the code. For this, we iterate over all chunks, and use the same parser as in the previous function to parse the headers one at a time.

Note that in this function, we have no pipelining whatsoever, however, the function `downloadPipelined()` gives a pipelined implementation of the function. We have another parameter now - the number of chunks we request for in a window. To do a pipelined implementation, we have three states - when we are not parsing

(i.e., when we are receiving), when we are in the header, and when we are in the body of a chunk. I have taken care of the fact that the received message can have a header starting in the middle of the received message, and that the message might have multiple headers if the buffer size is a bit more than the chunk size.

The parsing is slightly different from the previous function, where we were guaranteed that there is only one header and one body in whatever is in the buffer. Here we maintain the current chunk range, the message so far, the header so far, the message size, completed chunks and whether we are parsing or not.

4 Part 3

Problem: Make the downloader more efficient by using threading to open multiple connections and pipelining.

Solution: There are a few changes in the pipelined code in the previous part to make the pipelined code thread-safe; here I have implemented a chunk allocator which allocates a few chunks to a thread from a global pool of available chunks when it is called. Also I have used some locks to ensure that all operations are thread-safe. Some parameters are shifted to the thread arguments (for making it easier to call them from the function that creates all these threads). The rest of the code is pretty much the same and can be found in the function `downloadSingleThread()` for one thread, and the thread manager is the function `downloadMultipleConnections()`.

To implement the downloads from different servers, we first parse the URLs (which should contain the path of the file in the end) and then for the specified number of threads, use that information to download the chunks. To avoid hardcoding the file name and the URL, and hence the file size, we assume that the file actually exists at those paths and send a HEAD request to find the file size (we can check for file existence using a simple check which is done for the first server).

The graphs for some of the examined cases are attached towards the end.

1. **Does your download time keep decreasing with more and more parallel TCP connections?**

   Yes, it decreases with more parallel TCP connections. This happens because the idle time due to blocking calls is now occupied by threads that are free, and thus making the download more efficient.

2. **Are some connections faster than the others? Do some connections get stalled for some reason?**

   Within a certain amount of variance, all connections have roughly the same speeds and connections do not get stalled.

3. **Does your download time reduce if you download from different sources in parallel?**

   Keeping the total number of threads constant, the situation where we have more threads to a faster server has a better download speed. However, adding more threads of another server almost always reduces the download time (if the number of threads of the original server are kept the same).

4. **What does this tell you about where the bottlenecks lie?**

   In the case of the `vayu.iitd.ac.in` server, the connection is quite fast. Since python allows only one thread to run at a time, and there is not much effect of increasing the threads beyond a certain point, it seems that the RTT is not the bottleneck in the case of this server. However, for the `norvig.com` server, the RTT is the main bottleneck, since on increasing the number of threads, the performance goes up in a manner proportional to the number of threads, and the main reason where the threads reduce the time is when there is a blocking call, since python virtual machine runs a thread one at a time.

5. **Is one server faster than the other? Is your program able to use this to download more from the faster server**

   Yes, the `vayu.iitd.ac.in` server is faster than the `norvig.com` server, as can be seen from ping response times (8ms and 200+ms). The program is able to use this and since the total time taken by all threads is the same, we see that it roughly divides the load in the ratio of the per-thread download speeds.

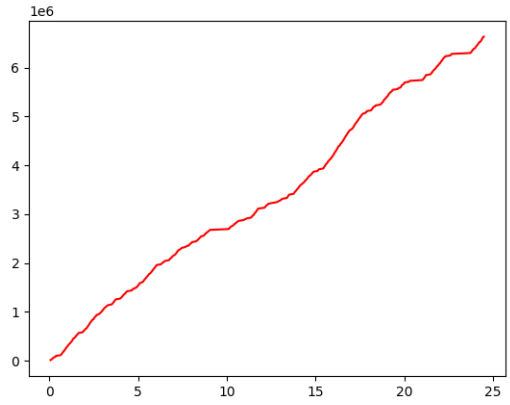
5 Part 4

Problem: Make this downloader more resilient by keeping a check for connection failures etc.

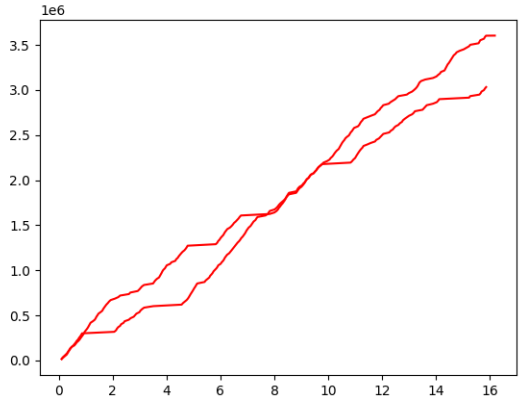
Solution: Now to make this resilient, we have to take care of disconnections, as well as errors in which the `recv` and `send` calls fail or connection is broken (in which case `recv` calls send an empty response). To ensure that these are taken care of, we enclose these calls in `try-except` blocks, and whenever they fail, we create a new socket and resend the remaining pipelined requests again.

6 Graphs for `vayu.iitd.ac.in`

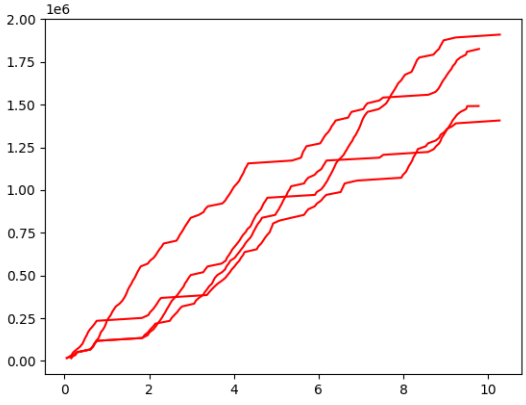
As can be seen below, the time decreases as we increase the number of TCP connections. The load shared by the threads is roughly equal, and the time taken by each thread to stop is also roughly the same.



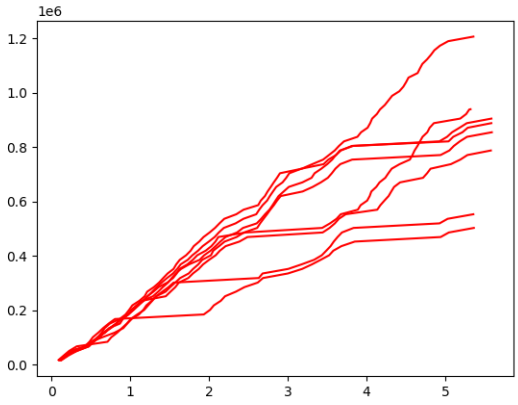
Bytes downloaded v/s time for 1 thread



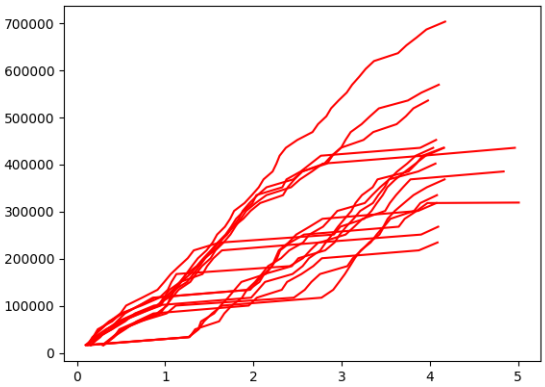
Bytes downloaded v/s time for 2 threads



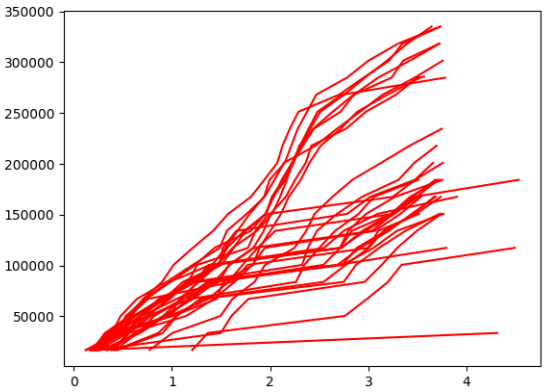
Bytes downloaded v/s time for 4 threads



Bytes downloaded v/s time for 8 threads



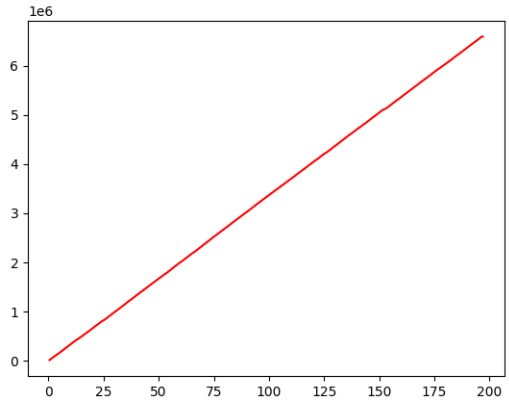
Bytes downloaded v/s time for 16 threads



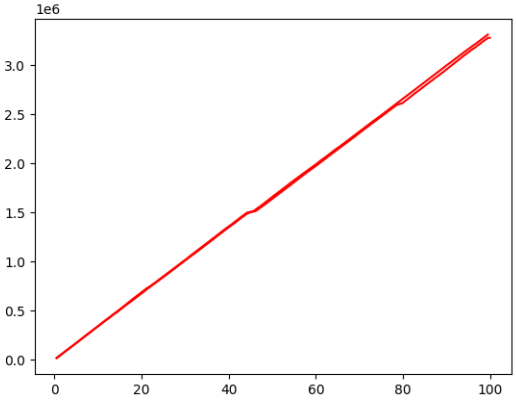
Bytes downloaded v/s time for 32 threads

7 Graphs for `norvig.com`

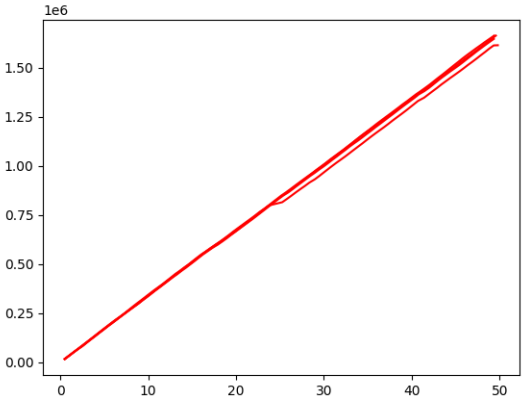
As for the previous server, here we observe a similar trend, but the time taken is almost inversely proportional to the number of threads. This happens since the RTT is the bottleneck here (as was seen on my connection, a ping to this server takes roughly 250ms, while that to the previous one takes 8ms). Most of the randomness is also gone, since the threads wait and finish waiting about the same time (there is less relative variance in the RTT for this server since the RTT is large, so threads are allocated more fairly).



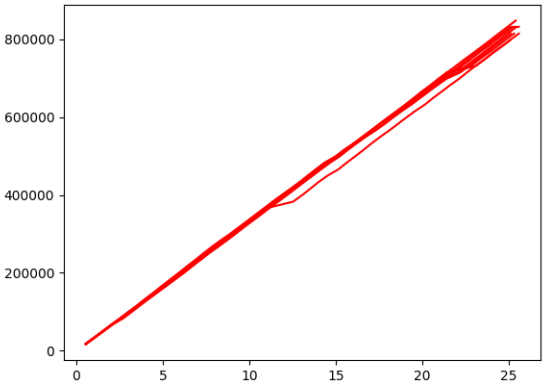
Bytes downloaded v/s time for 1 thread



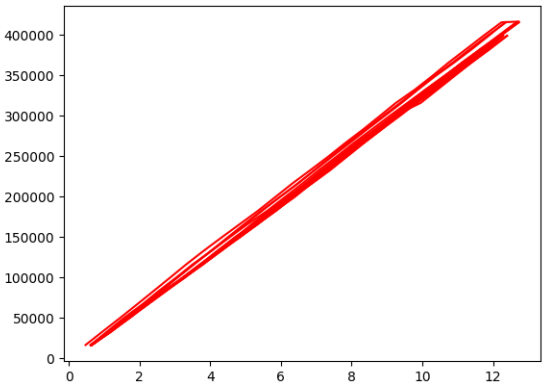
Bytes downloaded v/s time for 2 threads



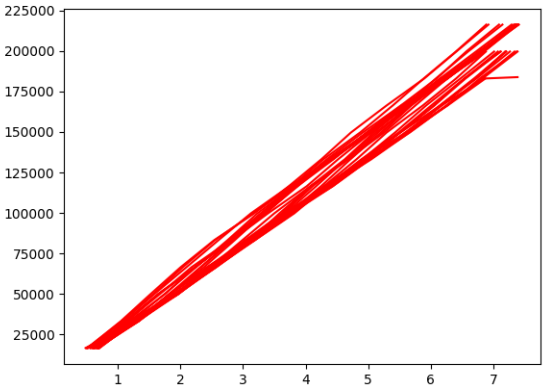
Bytes downloaded v/s time for 4 threads



Bytes downloaded v/s time for 8 threads



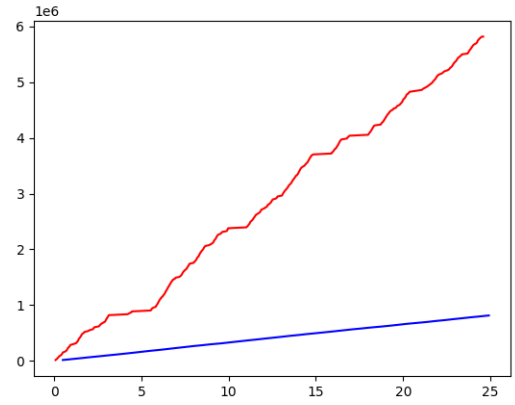
Bytes downloaded v/s time for 16 threads



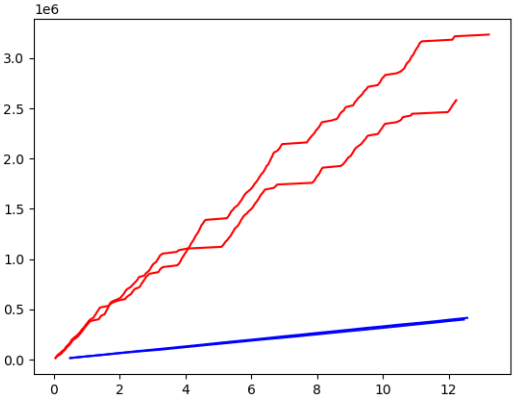
Bytes downloaded v/s time for 32 threads

8 Graphs for both servers at once

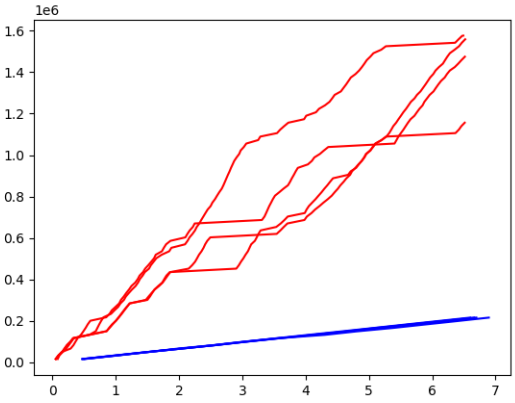
From various runs while changing the number of threads, by keeping the number of threads same for both servers, and then keeping the total number of threads constant, we can notice that the total size downloaded by the threads of a server is proportional to the speed of the server with that number of threads when comparing across servers. This shows that the distribution among threads is nearly optimal with respect to total time taken, and all the threads take the same time.
We also see that when we have the same total number of threads, the total download speed is higher when we have more threads for the faster server, as can be expected simply.



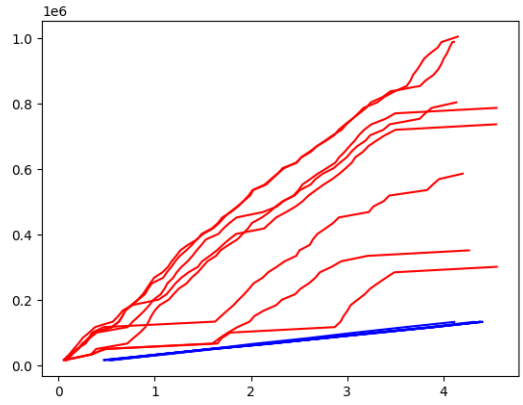
Bytes downloaded v/s time for 1 thread each



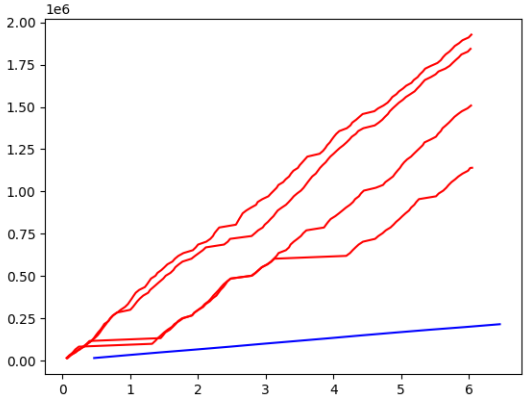
Bytes downloaded v/s time for 2 threads each



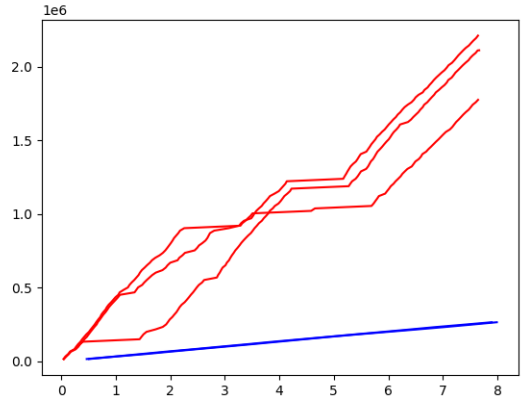
Bytes downloaded v/s time for 4 threads each



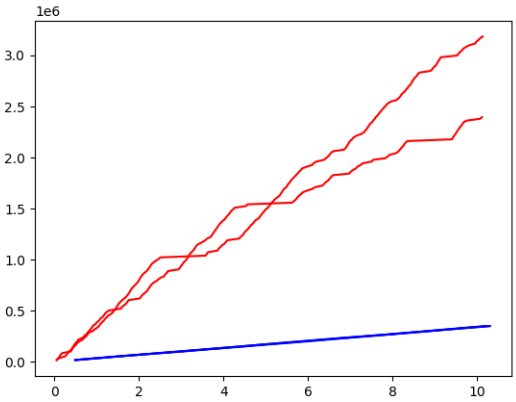
Bytes downloaded v/s time for 8 threads each



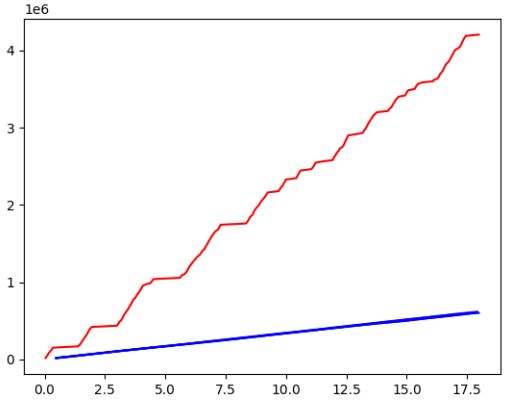
Bytes downloaded v/s time for 4, 1 threads on the first, second server respectively



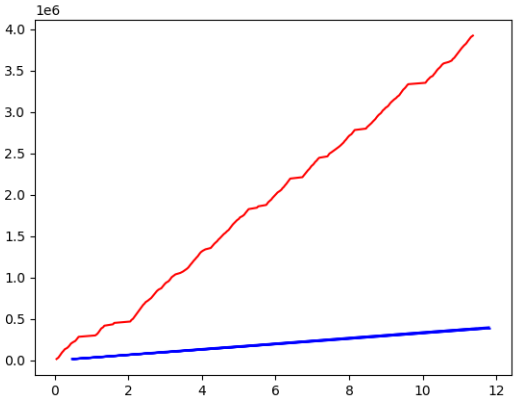
Bytes downloaded v/s time for 3, 2 threads on the first, second server respectively



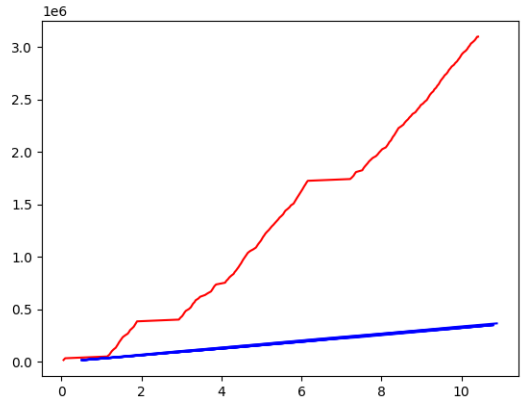
Bytes downloaded v/s time for 2, 3 threads on the first, second server respectively



Bytes downloaded v/s time for 1, 4 threads on the first, second server respectively



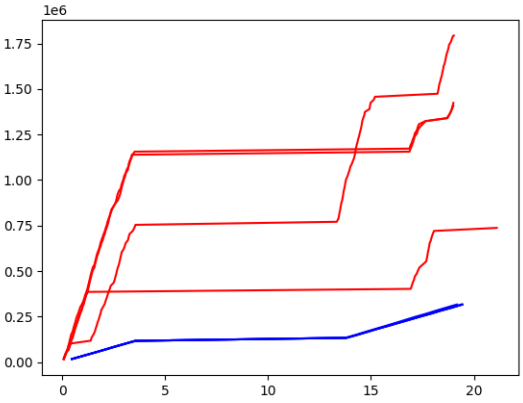
Bytes downloaded v/s time for 1, 7 threads on the first, second server respectively



Bytes downloaded v/s time for 1, 10 threads on the first, second server respectively

9 Graph for broken connections

As can be seen in the graph, broken connections do not fatally crash the download, and the download resumes when we come back online. Note that the alignment is not that clear, since we push the (timestamp, bytes downloaded) pairs only when we complete downloading a chunk, and we don't sample the downloaded bytes at a fixed interval of time. The connection timeout was set to be 2 seconds, and hence this also contributes to such an alignment.



Bytes downloaded v/s time for a connection broken due to disconnection from the internet