

# File Transfer and Priority Scheduling in Multipath QUIC

---

## Group Number - 10

<i>Saptarsi Saha</i>	<i>Manisha Jhunjunwala</i>	<i>Shailesh Navale</i>	<i>Madhur Navandar</i>
2020H1030109	2020H1030125	2020H1030169	2020H1030163

## Abstract

In the current scenario, QUIC(Quick UDP Internet Connection), developed by Google is being used extensively in Chrome to reduce page-load times, rebuffers and to have a low latency of client-server communication. It has replaced the traditional HTTPS/TLS/TCP stack and has included some cryptographic mechanisms to authenticate the server along with leveraging the reliability and congestion control mechanisms of modern TCP stacks. But it cannot exploit the multiple paths and several wireless interfaces existing between today's mobile devices. Here comes the multipath version of QUIC - MPQUIC which can take advantage of multiple paths existing between the two communicating hosts simultaneously resulting in increased utilisation of resources and efficiency. Although this protocol has not been implemented in the real-world setups, we have tried to simulate the working of MPQUIC over a mininet VM and have conducted some file transfer experiments using MPQUIC(built over UDP). Also, in the internet, web pages can be rendered as the multiple streams for e.g a web page might render the streams collecting the HTML text, video, audio, advertisement,etc. The implementation of the MPQUIC as presented in the Conext`17 research paper is not stream aware. So, we have studied the Priority Bucket scheduler([PStream](#)) and have presented our ideas to improvise on the time complexity of the scheduling algorithm from  $O(n^2)$  to  $O(n \log n)$ .

## 1. Introduction

In this paper we are presenting the implementation of the MPQUIC using GO language. We were able to implement the MPQUIC using the open-source QUIC-Go Repository by Quentin De Coninck. The first part of the project deals with the scenario of transferring/uploading a large file between the client and server in a multipath network setup. We have been successful in implementing the transfer of files of multiple sizes over different network conditions and characteristics(bandwidth, delay, losses) of the disjoint paths. We have also tested the resiliency of our multipath topology by dynamically making one of the interfaces down.

For the second part of the project, we have studied the Priority Bucket scheduler([PStream](#)) for the MPQUIC which schedules the data as streams in  $O(n^2)$  time complexity based on the priorities of the stream. By scheduling the streams based on the priority we reduce the response time for the web page rendering. We have tried to implement the PStream scheduler by referring to the research paper's source but could only proceed partially due to several certificate errors.

In the third part, we have analysed the existing PStream algorithm extensively and it seems that we can fill some missing gaps in it. So we have proposed a improvisation in the current algorithm and proved our idea to reduce the time complexity to  $O(n \log n)$ , with supporting calculations.

---

In this part we will have a quick look on how we can transfer files from host to server using MPQUIC. We have first tried to configure MPQUIC protocol in the real environment but was unable to do so because to support MPQUIC protocol we need to configure multiple tools to build and test our application as it is quite new and it's module is not yet deployed. We are focused on transmitting single files through multiple paths and to observe the difference in time it takes when we schedule a packet through a single path(as it was done previously in gQUIC) and through multipath QUIC.

## II. MPQUIC Setup

We have decided to go with the decision to use virtual machines to host all our test and then transfer the results. We could have gone with installing mininet directly in Ubuntu and proceed, but that would have needed a lot of tools to configure, build and test, because both QUIC and MPQUIC are still in development phase and are not built as modules yet.

We have decided to use Mininet VM available in the [CoNEXT 2017 Artifacts](#) for experimentation purposes. It comes bundled with Minitopo, a very useful helper tool to build multipath networks over Mininet.

## III. Simulating Networks

We decided to simulate a multi-path network using the network simulator tool mininet, which was installed inside a Linux virtual machine.

After installation of Mininet VM we first created the topology by using the mininet python scripts and/or minitopo sample.topo file which we designed to connect a host to a router using 2 paths.

Our created topology as following:

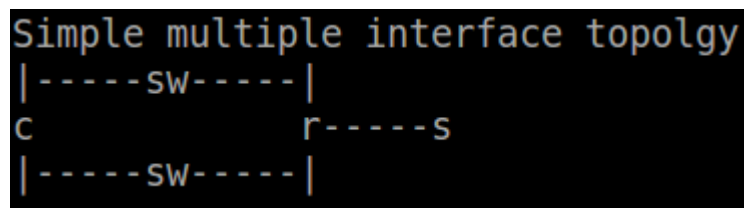


Fig 1. Two disjoint paths between Client and Server

The network has two hosts with client having two interfaces and server having one, and two switches linking the two hosts. The switches are then connected to a router which connects to the server. The IP addresses assigned to the hosts are as follows

### Client:

H1-eth0 : 10.0.0.1

H1-eth1 : 11.0.0.1

**Server** : 100.0.0.1

The topology creation was a little bit tricky because with our initial setup, the traffic was not evenly distributed between the 2 links. So we have made some tweaks to utilise all the paths equally.

**Minitopo to the rescue:** After careful consideration, we decided to move on to minitopo, which is a simple tool, based on mininet, to boot a simple network with n paths and run experiments between two hosts. We ran multiple experiments by tweaking the parameters in our sample.topo file as described in the later parts of the report.

#### IV. File Transfers over MPQUIC(UDP) protocol

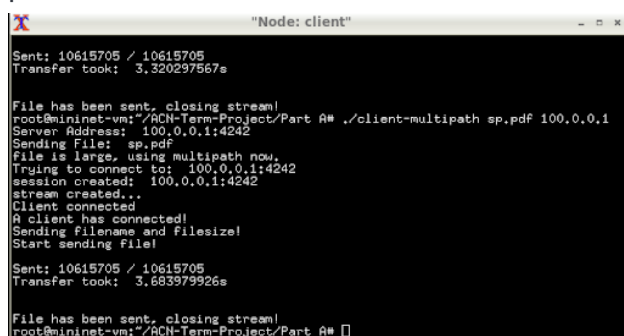
After successfully transferring files through QUIC we moved our focus to MPQUIC for which we required a parameter : **CreatePaths** which was missing in [lucas-clemente's QUIC-GO](#) implementation. We referred to the CoNEXT 2017 paper by Quentien De Conick which comes bundled with all the required tools such as minitopo and has provision to use this parameter too.

Steps that we performed while transferring file from client to server using multiple path:

1. Create a topology in mininet as in Fig 1.
  2. As quic-go implementation is only based on single path connection to transfer file packets between client and server in order to make it multi-path we needed to change the code a little bit and added the following snippet in both client and server go file.
- ```
quicConfig := &quic.Config { CreatePaths: true, }
```
3. The file transfer code for both the sides(client and server) - server.go and client.go files is firstly built inside mininet VM.
  4. The server and client is run in separate xterms.
  5. Server waits for the client to initiate a connection request. When the connection is established, the server decides whether to use multiple paths or not based on the file-size. We have provided a threshold in the server side, if the file size is bigger than the threshold of 10KB, multiple streams are created and transfer happens using MPQUIC.
  6. We have attached a storage folder for server while running the client so that the uploaded file gets stored there.

We observe that we were able to send the file successfully as shown in fig 2 & 3 below but utilization of both path is not efficient that we can see in fig 4.

So, how did we check that the tranfer was using MPQuic protocol only and not TCP. We checked the wireshark traces with these endpoints and confirmed that all these packets were using the UDP protocol.



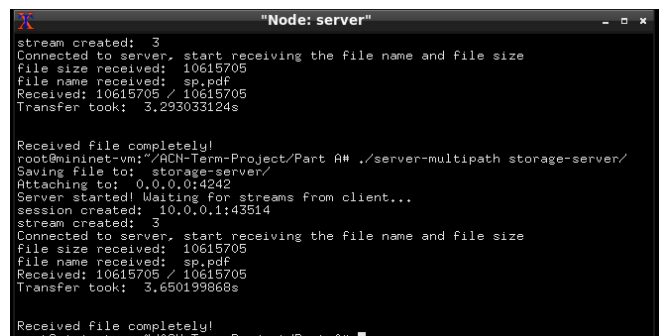
```
"Node: client"
Sent: 10615705 / 10615705
Transfer took: 3.320297567s

File has been sent, closing stream!
root@mininet-vm:~/ACN-Term-Project/Part A# ./client-multipath sp.pdf 100.0.0.1
Server Address: 100.0.0.1:4242
Sending File: sp.pdf
File is large, using multipath now.
Trying to connect to: 100.0.0.1:4242
session created: 100.0.0.1:4242
stream created...
Client connected
A client has connected!
Sending filename and filesize!
Start sending file!

Sent: 10615705 / 10615705
Transfer took: 3.683979926s

File has been sent, closing stream!
root@mininet-vm:~/ACN-Term-Project/Part A#
```

Fig 2: Client Xterm:



```
"Node: server"
stream created: 3
Connected to server, start receiving the file name and file size
File size received: 10615705
File name received: sp.pdf
Received: 10615705 / 10615705
Transfer took: 3.293033124s

Received file completely!
root@mininet-vm:~/ACN-Term-Project/Part A# ./server-multipath storage-server/
Saving file to: storage-server/
Attaching to: 0.0.0.0:4242
Server started! Waiting for streams from client...
session created: 10.0.0.1:43514
stream created: 3
Connected to server, start receiving the file name and file size
File size received: 10615705
File name received: sp.pdf
Received: 10615705 / 10615705
Transfer took: 3.650199868s

Received file completely!
root@mininet-vm:~/ACN-Term-Project/Part A#
```

Fig 3: Server Xterm:

| Ethernet - 4 |         | IPv4 - 4 |            | IPv6     |            | TCP - 3  |  | UDP - 4 |  |
|--------------|---------|----------|------------|----------|------------|----------|--|---------|--|
| Address      | Packets | Bytes    | Tx Packets | Tx Bytes | Rx Packets | Rx Bytes |  |         |  |
| 10.0.0.1     | 554     | 389 k    | 389        | 367 k    | 165        | 21 k     |  |         |  |
| 11.0.0.1     | 61      | 61 k     | 61         | 61 k     | 0          | 0        |  |         |  |
| 100.0.0.1    | 615     | 451 k    | 165        | 21 k     | 450        | 429 k    |  |         |  |
| 127.0.0.1    | 34      | 6,258    | 17         | 3,129    | 17         | 3,129    |  |         |  |

Fig 4: Uneven traffic through 2 paths

| Address   | Packets   | Bytes       | Tx Packets | Tx Bytes    | Rx Packets | Rx Bytes    |
|-----------|-----------|-------------|------------|-------------|------------|-------------|
| 10.0.0.1  | 546 495   | 384 361 964 | 364 753    | 370 270 231 | 181 742    | 14 091 733  |
| 100.0.0.1 | 1 307 856 | 923 002 282 | 433 824    | 33 486 730  | 874 032    | 889 515 552 |
| 127.0.0.1 | 70        | 12 274      | 35         | 6 137       | 35         | 6 137       |
| 11.0.0.1  | 761 361   | 538 640 318 | 509 279    | 519 245 321 | 252 082    | 19 394 997  |

Fig 5: Traffic between 2 paths is evenly distributed(appx.)

In order to efficiently use both the paths in topology, we changed the code in [config-client.sh](#) file

```
- route add default gw 10.0.0.2 client-eth0
+ ip route add default scope global nexthop via 10.0.0.2 dev client-eth0
```

We tried running the same experiment with a sample minitopo file and utilizing 4 channels for communication and got the same approx result whereby the packets were being divided equally amongst the channels and the links were being used effectively.

|           |        |            |       |           |       |            |   |
|-----------|--------|------------|-------|-----------|-------|------------|---|
| 10.0.0.1  | 3 916  | 2 778 258  | 2 549 | 2 668 636 | 1 367 | 109 622    | - |
| 10.1.0.1  | 15 052 | 10 735 485 | 5 226 | 410 999   | 9 826 | 10 324 486 | - |
| 127.0.0.1 | 32     | 5 594      | 16    | 2 797     | 16    | 2 797      | - |
| 10.0.1.1  | 3 809  | 2 748 144  | 2 498 | 2 645 941 | 1 311 | 102 203    | - |
| 10.0.2.1  | 3 788  | 2 741 057  | 2 499 | 2 640 772 | 1 289 | 100 285    | - |
| 10.0.3.1  | 3 539  | 2 468 026  | 2 280 | 2 369 137 | 1 259 | 98 889     | - |

Fig 6: Server IP is 10.1.0.1(in violet) and the client is connected to 4 interfaces(10.0.0.1,10.0.1.1,10.0.2.1,10.0.3.1) in red.

Initial bwm-ng results also confirm the same.

| iface    | Rx           | Tx           | Total        |
|----------|--------------|--------------|--------------|
| s0-eth1: | 613.26 KB/s  | 21.50 KB/s   | 634.76 KB/s  |
| s3-eth1: | 614.84 KB/s  | 21.73 KB/s   | 635.59 KB/s  |
| s1-eth2: | 21.49 KB/s   | 611.23 KB/s  | 632.39 KB/s  |
| s2-eth1: | 700.13 KB/s  | 22.06 KB/s   | 719.70 KB/s  |
| lo:      | 105.12 KB/s  | 105.12 KB/s  | 210.24 KB/s  |
| s1:      | 68.01 KB/s   | 0.00 KB/s    | 68.01 KB/s   |
| s0-eth2: | 21.50 KB/s   | 610.42 KB/s  | 631.36 KB/s  |
| s3-eth2: | 21.73 KB/s   | 610.57 KB/s  | 631.44 KB/s  |
| s1-eth1: | 611.23 KB/s  | 21.49 KB/s   | 632.72 KB/s  |
| s2-eth2: | 22.06 KB/s   | 610.98 KB/s  | 631.77 KB/s  |
| s0:      | 2.16 KB/s    | 0.00 KB/s    | 2.16 KB/s    |
| s3:      | 68.28 KB/s   | 0.00 KB/s    | 68.28 KB/s   |
| s2:      | 68.29 KB/s   | 0.00 KB/s    | 68.29 KB/s   |
| eth0:    | 0.18 KB/s    | 0.35 KB/s    | 0.53 KB/s    |
| total:   | 2536.60 KB/s | 2526.37 KB/s | 5060.53 KB/s |

## V. Checking the resiliency of the multipath network

As the concept of multi-path network says, that if any path or stream goes down due to changing network connections, there should not be any hindrance with the connection if atleast one path is up. All the traffic should get distributed in the left-over paths which are active.

But we discovered that our topology was not able to handle it at first. We tried several ways to enable our code to make this choice dynamically but we faced many challenges in making this setting. We tried to tinker with the topology but discovered that there was some issue with the mininet which didn't allow us this functionality. So whenever we made any path down, we got a segmentation fault. We thought over it and decided to test in another way, that whether our code needed any change or it was really an issue with the mininet VM setup.

We configured 2 mininet VM's, one for client and another for server. Server having one interface, but for the client side, we tried to add another interface by making changes to the NAT settings.

We then performed the whole experiment of transferring file again, this time by adding 2 other steps.

1. We made one interface down before running the client.
2. We made one interface down between the transfer on the fly.

We were surprised to see the results. So with the same client and server code and the topo setup, we could dynamically transfer the traffic to the left over paths if we made any interface down.

The wireshark traces confirmed the same.

| Address         | Packets | Bytes       | Tx Packets | Tx Bytes    | Rx Packets | Rx Bytes    |
|-----------------|---------|-------------|------------|-------------|------------|-------------|
| 192.168.221.145 | 160 601 | 117 317 026 | 54 996     | 4 229 148   | 105 605    | 113 087 878 |
| 13.126.27.131   | 6       | 552         | 3          | 276         | 3          | 276         |
| 103.134.252.11  | 6       | 552         | 3          | 276         | 3          | 276         |
| 5.189.141.35    | 4       | 368         | 2          | 184         | 2          | 184         |
| 91.189.89.198   | 4       | 368         | 2          | 184         | 2          | 184         |
| 192.168.221.150 | 160 581 | 117 315 186 | 105 595    | 113 086 958 | 54 986     | 4 228 228   |

Fig 7:

| Address         | Packets | Bytes       | Tx Packets | Tx Bytes   | Rx Packets | Rx Bytes    |
|-----------------|---------|-------------|------------|------------|------------|-------------|
| 192.168.221.146 | 58 710  | 42 804 471  | 38 802     | 41 290 711 | 19 908     | 1 513 760   |
| 192.168.221.145 | 161 232 | 117 466 822 | 54 554     | 4 145 701  | 106 678    | 113 321 121 |
| 192.168.221.147 | 102 522 | 74 662 351  | 67 876     | 72 030 410 | 34 646     | 2 631 941   |

Fig 8:

*Fig 7. One interface is down and the traffic is routed through the other interface. Server IP is 192.168.221.145 & client is 192.168.221.150*

*Fig 8. We cut interface with IP 192.168.221.146 dynamically while file transfer is happening and can see that the traffic gets routed through the other interface.*

## VI. Challenges Faced

This project was not short of any challenges, from modifying the QUIC-go code to run for MPQUIC to changing the topology files for different scenarios and getting stuck at the expired certificates, we faced it all. Some of them are listed below:

### 1. Implementing file-transfer in QUIC.

While using quic-go client the method was set to use get as default. We modified it to use post method and implemented the quic-protocol.

```
- rsp, err := hclient.Get(addr)
+ rsp, err := hclient.Post(addr, "binary/octet-stream", file)
```

### 2. Multipath Next!

To implement multipath-quic we had to make changes to the existing code and include the

`quicConfig := &quic.Config{ CreatePaths: true, }` parameter. This parameter is not accessible by the quic-go module and we had to fall-back to MPQUIC-VM.

### 3. Design Changes Finally.

We were using 2 paths and the traffic ratio was divided in 40:1.

We found critical issues in our topology and made changes in the client-mininet topology to bring down the traffic ratio to 1:1 (approximately).

## VII. Testing with different network conditions

We have used tools like **Netem** and **bwm-ng** to modify the bandwidth, delay and losses of all the paths and observe the results.

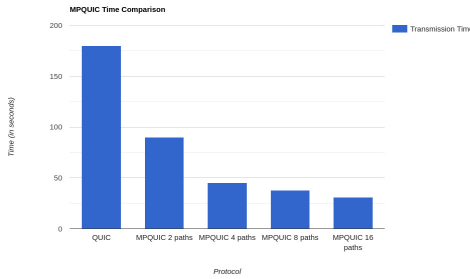


Fig: Number of Paths versus Time comparison

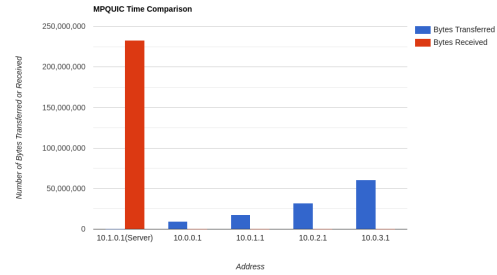


Fig: Bandwidths versus transferred bytes

So, in Fig 1: We can observe that MPQUIC has taken a considerably less time than QUIC for large file transfers. Also, when we increased the number of paths, the transmission time kept on decreasing.

In Fig 2, we have configured the 5 interfaces of client with bandwidths in the ratio of 1:2:4:8:16. We can see that on paths with higher bandwidths, number of bytes transferred was more.

## PART B: Priority stream bucket

For this part, we have read and presented the paper on [4][PriorityBucket: A Multipath-QUIC Scheduler on Accelerating First Rendering Time in Page Loading](#). We have also tried to implement it but faced some difficulties which are mentioned below.

The response time of a web page can be the time a web server takes to show the first visual feedback to the user. Visual feedback is the HTML, CSS, or Javascript components that make up the webpage. So the idea is to prioritize such streams that make up the front-end design of the web page over the other streams such as advertisement.

Protocols like HTTP 1.1 or HTTP 2.0 suffer from the problem of HOL blocking protocol since they were using TCP. To remove the unnecessary blockage of independent requests, QUIC protocol multiplexes HTTP requests/responses over a single UDP connection. With UDP protocol QUIC ensures that late packets from each individual stream will not block other concurrent streams which resolves HOL blocking problems.

In this project we will focus on an algorithm which deals with scheduling a stream to different paths based on path bandwidth or number of higher priority streams in the path and number of equal priority streams in a path compared to a stream to be scheduled by using MPQUIC protocol. The idea behind such scheduling of the packets over different paths such that the completion time of the stream over each scheduled path is equal. In this way we are trying to optimize the completion time and complete the transfer in more efficient manner.

As show in the below figure the buckets with lower number will have the higher priority. Inside the bucket scheduling will take place in round robin fashion.

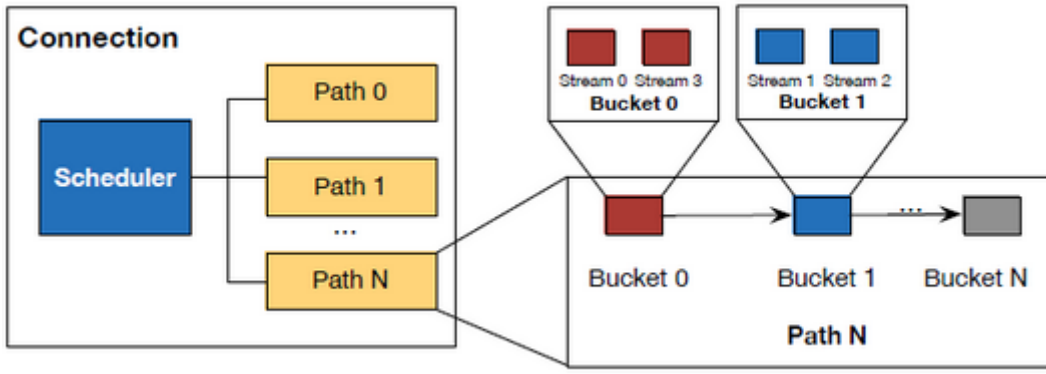


Fig 1: Structure of priority bucket design [4]

The streams with higher priority can preempt the stream with lower priority during the transfer.

---

**Algorithm 1** PriorityBucket algorithm

---

```

1: procedure SCHEDULESTREAM
2:    $sortedList \leftarrow$  sort paths by ascending order of  $r_i$ 
3:   initialize  $d_i = 0, i \in [1, 2, \dots, n]$ 
4:    $D' \leftarrow D$ 
5:   for path  $i$  in  $sortedList[1, \dots, n - 1]$  do
6:      $r_{gap} \leftarrow r_{i+1} - r_i$ 
7:     if  $r_{gap} \neq 0$  and  $D' > 0$  then
8:       if  $D' \geq r_{gap} \times \sum_{k=1}^i b_k$  then
9:         for  $j \leftarrow 1$  to  $i$  do
10:           $inc \leftarrow r_{gap} \times b_j$ 
11:           $d_j \leftarrow d_j + inc$ 
12:           $D' \leftarrow D' - inc$ 
13:        end for
14:      else
15:        for  $j \leftarrow 1$  to  $i$  do
16:           $inc \leftarrow D' \times (b_j / \sum_{k=1}^i b_k)$ 
17:           $d_j \leftarrow d_j + inc$ 
18:        end for
19:         $D' \leftarrow 0$ 
20:        break
21:      end if
22:    end if
23:  end for
24:  if  $D' > 0$  then
25:    for path  $i$  in  $sortedList$  do
26:       $d_i \leftarrow d_i + D' \times (b_i / \sum_{k=1}^n b_k)$ 
27:    end for
28:  end if
29: end procedure

```

**Stream scheduling approach:**

Below table, represents the symbols and meanings of the terms used in this algorithm.



**Table 1: Scheduling Parameters**

| Symbol | Meaning                                           |
|--------|---------------------------------------------------|
| $D$    | total data volume of stream                       |
| $h_i$  | total data volume of streams with higher priority |
| $e_i$  | total data volume of streams with equal priority  |
| $T$    | overall completion time of stream                 |
| $i$    | path $i$                                          |
| $n$    | number of paths                                   |
| $o_i$  | estimated one-way delay of the path $i$           |
| $b_i$  | estimated bandwidth of the path $i$               |
| $d_i$  | fraction of data assigned to path $i$             |
| $t_i$  | completion time of stream $s$ on path $i$         |

Along with the parametrers mentioned in table 1, algorithm introduces one more parameter  $r_i$ .

$$r_i = \frac{h_i + e_i}{b_i} + o_i$$

$\therefore r_i$  = time required for transmission of equal and higher priority streams + one time delay.

i.e the time required for completion if we do not send any data over path  $i$ .

It can be divided in the below steps:

### Step 1:

Sort the paths in ascending order of  $r_i$ .

### Step 2:

The fillgap process schedules data to balance the completion time gap of multiple paths. It fetches two paths with smallest  $r_i$  and will try to schedule balance out the time.e.g. At first the algorithm will consider path1 and path2 with smallest  $r_i$  i.e.  $r_1$  and  $r_2$ .

We want to find  $d_1$  such that it satisfies below equation.

$$r_1 + \frac{d_1}{b_1} = r_2 \text{ (here } b_1 \text{ is the bandwidth of path1 )}$$

On solving,

$$d_1 = (r_2 - r_1) \times b_1$$

Similarly this process will be repeated until all the data is allocated or completion time of the paths becomes equivalent. This process is called as gap filling process in research paper.

### Step 3

If any data is still not have been scheduled, schedule it in the ratio of the bandwidths of the paths.

### Implementation Attempt

In order to implement this research paper, we have tried to use already [implemented github](#) repo from one of the author Xiang-shi. We PStream repository in the local virtual machine and tried to set it up.



Initially, the repo was not able to run due to the path issues. In order to implement this we had to clone this repository in the repository used implementation of Part A. We also had to change the name of this repository from 'PStream' to 'pstream'.

When we tried to run this we have faced the few errors which we were not able to resolve yet.

```
mininet@mininet-vm: ~/go/src/github.com/lucas-clemente/pstream/example/client
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example/client$ cd client
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example/client$ ls
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example/client$ g
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example/client$ g
p run main.go https://localhost:6121/demo/tiles
GET https://localhost:6121/demo/tiles
Starting new connection to localhost (0.0.0.0:40494 -> 127.0.0.1:6121), connect
ionID e3280ac1ea2f6bca, version 512
ScheduleMultiplePaths():
stream 1: size 0, priority 6(0 255 %d(bool=false))
path 0: bandwidth %d(float64=0) Mbps, rtt 0s
assigned to path 0
Certificate validation failed: x509: certificate has expired or is not yet vali
d
Closing session with error: ProofInvalid
Connection e3280ac1ea2f6bca closed.
got response for https://localhost:6121/demo/tiles: (*http.Response)(nil)
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x40 pc=0x6d9227]

goroutine 5 [running]:
main.main.func1(0xc42006f260, 0xc420012600, 0x7ffdb48275cf, 0x21)
/home/mininet/go/src/github.com/lucas-clemente/pstream/example/client/m
ain.go:64 +0x107
created by main.main
/home/mininet/go/src/github.com/lucas-clemente/pstream/example/client/m
ain.go:53 +0x332
exit status 2
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example/client$
```

Fig 2:Pstream Client

```
mininet@mininet-vm: ~/go/src/github.com/lucas-clemente/pstream/example
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example$ ls
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example$ ls
aes12 fnv128a pstream quic-clients quic-go quic-go-certificates
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example$ cd pstream/
mininet@mininet-vm:~/go/src/github.com/lucas-clemente/pstream/example$ go run ma
in.go
2021/01/28 21:14:37 Jan 28 21:14:37.880080
Serving new connection: e3280ac1ea2f6bca, version 512 from 127.0.0.1:40494
2021/01/28 21:14:37 Jan 28 21:14:37.880339 ScheduleMultiplePaths():
2021/01/28 21:14:37 Jan 28 21:14:37.880356 stream 1: size 0, priority 6(0 255 %
%d(bool=false))
2021/01/28 21:14:37 Jan 28 21:14:37.880364 path 0: bandwidth %d(float64=0) Mbp
s, rtt 0s
2021/01/28 21:14:37 Jan 28 21:14:37.880370 assigned to path 0
2021/01/28 21:14:38 Jan 28 21:14:38.756595 Closing session with error: ProofInv
alid:
```

Fig 3:Pstream Server

As show in the figure 2 and 3, we faced the issue with the certificate and we will try to resolve this in future.

## PART C: Proposed Improvement on Priority Bucket Algorithm

Below is the proposed algorithm as an improvement for the algorithm 1.

We have reduced the complexity of this algorithm to  $O(n \log n)$ .

### Algorithm 2 Optimal Priority Bucket

- 1: **Procedure:** OptimalScheduleStream
- 2:  $\text{sortedList} \leftarrow$  sort the paths by ascending order of  $r_i$
- 3: initialize  $d_i = 0, i \in [1, 2, \dots, n]$
- 4:  $D' \leftarrow D$
- 5:  $b_{sum} = 0$
- 6:  $opt = 0$
- 7:  $D_{sum} = 0$
- 8: **for**  $k = 2$  to  $n$  **do**
- 9:  $b_{sum} \leftarrow b_{sum} + b_{k-1}$
- 10:  $d_{sum} \leftarrow d_{sum} + r_{k-1} * b_{k-1}$
- 11:  $d_{cal} \leftarrow r_k * b_{sum} - d_{sum}$
- 12: **if**  $d_{cal} = D'$  **then**
- 13:  $opt = k$
- 14: **break**
- 15: **else if**  $d_{cal} > D'$  **then**
- 16:  $opt \leftarrow k - 1$
- 17: **break**
- 18: **else**

```

19:      $opt = k$ 
20:   end if
21: end for
22: for  $i = 1$  to  $opt - 1$  do
23:    $d_i = (r_{opt} - r_i) * b_i$ 
24:    $D' \leftarrow D' - d_i$ 
25: end for
26: if  $D' > 0$  then
27:    $b_{opt} \leftarrow opt$ 
28:   for  $j \leftarrow 1$  to  $n - opt$ 
29:     if  $r_{opt} + \frac{D'}{\sum_{i \in [1, opt+j]} b_i} < r_{opt+j}$  then
30:       break
31:     end if
32:    $b_{opt} \leftarrow k + j$ 
33: end for
34: for  $i \leftarrow 1$  to  $b_{opt}$  do
35:    $d_i \leftarrow d_i + \frac{D' * b_i}{\sum_{k \in [1, b_{opt}]} b_k}$ 
36: end for
37: end if

```

---

In the original algorithm, we have observed that unnecessary computation is being performed on the line 7 to 12 since the algorithm do not know the optimal paths. Hence each time a new path is being tested we are filling the gaps between all the paths.

As all the values necessary to compute are already present we can hence reduce the time complexity of this algorithm to  $O(n \log n)$ .

The algorithm can be divided in the logical steps as follows:

### Step 1: Finding the optimal number of the paths before scheduling (line 1 to 21)

1. As explained in the part B, we will first calculate the  $r_i$  values for each path and sort them in ascending order.
2. In order to reduce the complexity, we will try to find the optimal path of the algorithm by using the below equation so that the fill gap process is not required to be repeated.

Assume that first  $opt$  paths are required for scheduling the data such that it follows below equation. Here the terminologies have the same meaning as mentioned in the Table 1.

$$\begin{aligned}
 d_1 + d_2 + \dots + d_{opt} &\leq D \\
 \therefore \sum_{i \in [1, opt]} (r_{opt} - r_i) * b_i &\leq D \\
 \therefore r_{opt} * \sum_{i \in [1, opt]} b_i - \sum_{i \in [1, opt]} r_i b_i &\leq D \quad (1)
 \end{aligned}$$

We will try to find the maximal value of  $opt$  which satisfy the equation 1.

This can be performed in  $O(n)$  steps and sorting can be performed in  $O(n \log n)$  steps.

### Step 2: Scheduling the calculated data (line 21 to 25)

1. We will get the value of  $opt$  by performing the step 1 of the algorithm.
2. We will schedule the data  $d_i$  on the path  $i$  such that it satisfies the below equation.

$$d_i = (r_{opt} - r_i) * b_i$$

3. We will also calculate the data  $D'$  which is yet to be scheduled by below equation.

$$D' = D - \sum_{i \in [1, opt]} d_i$$

4. This will take  $O(n)$  steps.

### Step 3: Scheduling the remaining data (line 26 to 36)

1. If there is a data which is yet to be scheduled we will try to schedule it across the paths in the ratio of the bandwidth such that it does not increase the delay more than  $b_{opt}$  path.
2. By the end of the step 2 each path from 1 to  $opt$  will have the delay as  $r_{opt}$ .
3. Now we are trying to schedule the remaining data in the ratio of the bandwidth of the paths. But it is possible that paths after  $opt$  i.e.  $opt+1, opt+2$  have the higher bandwidth and overall time delay becomes less if we schedule some of the data on it.
4. Hence we will try to find maximum value of  $b_{opt}$  i.e.  $(opt + j)$  which satisfies the below equation.

$$r_{opt} + \frac{D'}{\sum_{i \in [1, opt+j]} b_i} < r_{opt+j}$$

5. We will then segregate the data in the ratio of the bandwidths on the paths 1 to  $b_{opt}$ . This step requires  $O(n)$  time.

Hence overall the time complexity of the algorithm is reduced from  $O(n^2)$  to  $O(n \log(n))$ . We will try to implement this practically in future.

### References:

1. <https://multipath-quic.org/2017/12/09/artifacts-available.html>
2. <https://github.com/qdeconinck/mp-quic>
3. <https://github.com/lucas-clemente/quic-go>
4. [PriorityBucket: A Multipath-QUIC Scheduler on Accelerating First Rendering Time in Page Loading](#)
5. <https://github.com/Xiang-Shi/PStream>