

Review of ML application for Multipath QUIC

BY

Name of the Student

GHANDIKOTA SRIKRISHNA

ID Number

2020H1030112H

**Under the supervision of
Prof. Paresh Saxena**

BITS G540 RESEARCH PRACTICE



**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, HYDERABAD CAMPUS
February 2021**

Table of Contents

1. Aim and Initial Methodology
2. Background Work
3. File transfer using MPQUIC over Mininet
4. File transfer using MPQUIC over a Router
5. Observations
6. Future Work
7. References

Aim and Initial Methodology

Understanding multipath protocols, QUIC and MPQUIC. Analyzing different types of MPQUIC schedulers. Implementing file transfer between 2 computers over 2 paths using MPQUIC with the use of quic-go, a QUIC implementation in pure Go. Analyzing the effect of different metrics applied over the path.

Background Work

QUIC is an encrypted, multiplexed, and low-latency transport protocol designed from the ground up to improve transport performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms. Multipath QUIC or MPQUIC is an extension to the QUIC protocol that enables hosts to exchange data over multiple networks over a single connection. There have been improvements in schedulers used in MPQUIC such as Peekaboo, a novel learning-based multipath scheduler that is aware of the dynamic characteristics of the heterogeneous paths.

Literature Review

De Coninck and Bonaventure (2017) talks developing multi-path for QUIC protocol developed by google. QUIC combines the functionalities of TCP,HTTP/2 and TLS over UDP without any other protocol. Adding multipath features to a system like QUIC has two major objectives. The first is to combine resources from several pathways to transmit data over a single connection. The Second one being to withstand network outages. One wireless network can fail at any time on dual interface hosts with wireless interfaces, such as smartphones, and users expect their apps to move to the other without any impact. The main functions of MPQUIC are Path Identification, Reliable Data Transmission, Path Management, Packet Scheduling and Congestion Control.

Wu et al. (2020) talk about a learning-based multipath scheduler that takes into account the dynamic features of diverse routes called Peekaboo. By constantly interacting with its surroundings, an Reinforcement agent attempts to learn the policy that maximises the cumulative reward. The congestion control problem, for example, may be seen as a decision-making problem, and RL algorithms have been demonstrated to give significant improvements. Instead of broadcasting the packet based on the RTT values, when both pathways are available, the scheduler learns how to transmit the packet over each path. The RL is trained on these four variables: sender window size, current window size, RTT and number of packets. The reward is throughput that is bytes per second. Peekaboo outperforms the current MPQUIC schedulers, with performance gains of up to 31.2 percent in simulated networks and up to 36.3 percent in real-world settings.

File transfer using MPQUIC over Mininet

Requirements:

1. OS == Linux
2. Git
3. go lang version >= 1.9.2(Preferably 1.9.2)
4. Mininet version == 2.3.0

Note: The entire procedure is done in Ubuntu 20.04

Installing git:

- Enter the following command to install **git** in your system.

```
sudo apt-get install git
```

Installing go lang:

- Download the tar file of go(1.9.2) from the below link:

<https://golang.org/doc/install?download=go1.9.2.linux-amd64.tar.gz>

- After downloading enter the following commands to install go:

```
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.9.2.linux-amd64.tar.gz
```

```
export PATH=$PATH:/usr/local/go/bin > ~/.profile
```

```
source ~/.profile
```

```
go version
```

- The output should be:

```
go version go1.9.2 linux/amd64
```

Installing Mininet:

- **Installing requirements for Mininet:**

- Installing python2 with pip

```
sudo apt install python-pip
```

sudo apt install python

- Enter the following commands to install mininet:

git clone git://github.com/mininet/mininet

cd mininet

git tag #list available versions

git checkout -b mininet-2.3.0 2.3.0

cd ..

mininet/util/install.sh -a

- For testing the installation of mininet enter the following command:

sudo mn --test pingall

This command will run a test topology. If it runs without any errors, mininet is successfully installed.

Installing MPQUIC:

- We use quic-go, a QUIC implementation in go lang to create a server, a client and run it on a mininet topology with multipath to test our implementation of MPQUIC.
- Enter the following commands to install MPQUIC over quic-go:

#Get upstream quic-go code

go get github.com/lucas-clemente/quic-go

#Add the mp-quic remote and checkout to the prototype branch

cd ~/go/src/github.com/lucas-clemente/quic-go

git remote add mp-quic https://github.com/qdeconinck/mp-quic.git

git fetch mp-quic

git checkout conext17

#Download all dependency packages

go get -t -u ./...

#If this code results in a Handshake error, enter the following commands

cd ~/go/src/github.com/bifurcation/mint

git reset --hard a6080d464fb57a9330c2124ffb62f3c233e3400e

#Check that no more build issues arise

cd ~/go/src/github.com/lucas-clemente/quic-go

go build

#This should not produce any output

Code for setting up the topology:

- Extract the attached zip file into home directory and open terminal in same directory
- Setup the mininet topology by running the following file:

sudo python topo.py

- This starts the mininet topology. Enter the following command to open client and server xterms:

mininet> xterm server client

- Open the server xterm and enter the following:

./server/main -www . -certpath ~/go/src/github.com/lucas-clemente/quic-go/example/ -bind 0.0.0.0:6121

- Open the client xterm and enter the following:

cd client

./main1 -m <https://100.0.0.1:6121/a.tar.gz>

#a.tar.gz is replaced with a different file name as long as the same file exists in the server folder.

- File transfer is done.

File transfer using MPQUIC over a Router

Requirements:

- A wifi router with at least 2 LAN connections.
- 2 Computers with MPQUIC installed. (Follow the above instructions)
- A LAN cable.

Aim: To transfer files from one computer to the other using MPQUIC

Setup

- Connect both the computers to the router over WiFi using ssid and password.
- Here one computer acts as a client and the other as the server.
- We would like to have multiple paths at the client side, so connect the client computer to the router using a LAN cable.

Code Setup(Server side):

- Create a folder *server_files* in the following directory:
`~/go/src/github.com/lucas-clemente/quic-go/`
- Copy the file you would like to transfer to the client into the *server_files* folder.
- Run the following command inside *server_files*:
`cp ~/go/src/github.com/lucas-clemente/quic-go/example/main.go .`
- Run the server using the following commands:
`go build main.go`
`./main -www . -certpath ~/go/src/github.com/lucas-clemente/quic-go/example/ -bind 0.0.0.0:6121`

Code Setup(Client side):

- The interfaces in the connection manager of mpquic are defaulted to mininet interfaces.
- First find out the interfaces on your client machine using *ifconfig -s*. The output is as follows:


```
kree@kree-pc: ~  
kree@kree-pc:~$ ifconfig -s  
Iface      MTU      RX-OK RX-ERR RX-DRP RX-OVR      TX-OK TX-ERR TX-DRP TX-OVR Flg  
enp3s0     1500     2308746 0      0 0      1398712 0      0      0 BMRU  
lo         65536    125204 0      0 0      125204 0      0      0 LRU  
wlo1       1500     88958 0      0 0      3658 0      0      0 BMRU  
kree@kree-pc:~$
```

- We can see that enp3s0 and wlo1 are the LAN and WiFi interfaces respectively.
- We must add these interfaces in the connection manager where paths are set up according to the interfaces.
- The connection manager is pconn_manager.go which can be found in the following directory.

~/go/src/github.com/lucas-clemente/quic-go/pconn_manager.go

- In createPconns() function, replace line 205 with the following:

```
if !strings.Contains(i.Name, "eth") && !strings.Contains(i.Name, "rmnet") && !strings.Contains(i.Name, "wlan") &&  
!strings.Contains(i.Name, "enp3s0") && !strings.Contains(i.Name, "wlo1"){
```

- Create a folder client_files in the following directory:

~/go/src/github.com/lucas-clemente/quic-go/

- In client_files, create a file client.go and copy the following code:
- The client side code is as follows:

```
package main
```

```
import (
```

```
    "bytes"
```

```
    "crypto/tls"
```

```
"flag"  
"io"  
"log"  
"net/http"  
"os"  
"sync"  
"time"  
"fmt"  
"strings"  
"io/ioutil"  
  
quic "github.com/lucas-clemente/quic-go"  
"github.com/lucas-clemente/quic-go/h2quic"  
"github.com/lucas-clemente/quic-go/internal/utils"  
)
```

```
func main() {  
    verbose := flag.Bool("v", false, "verbose")  
    multipath := flag.Bool("m", false, "multipath")  
    output := flag.String("o", "", "logging output")  
    cache := flag.Bool("c", false, "cache handshake information")
```

```
flag.Parse()

urls := flag.Args()

fmt.Println("urls", urls)

if *verbose {

    utils.SetLogLevel(utils.LogLevelDebug)

} else {

    utils.SetLogLevel(utils.LogLevelInfo)

}

if *output != "" {

    logfile, err := os.Create(*output)

    if err != nil {

        panic(err)

    }

    defer logfile.Close()

    log.SetOutput(logfile)

}

quicConfig := &quic.Config{

    CreatePaths: *multipath,

    CacheHandshake: *cache,

}
```

```
hclient := &http.Client{  
    Transport: &h2quic.RoundTripper{QuicConfig: quicConfig, TLSClientConfig:  
&tls.Config{InsecureSkipVerify: true}},  
}
```

```
var wg sync.WaitGroup
```

```
wg.Add(len(urls))
```

```
for _, addr := range urls {
```

```
    utils.Infof("GET %s", addr)
```

```
    go func(addr string) {
```

```
        start := time.Now()
```

```
        rsp, err := hclient.Get(addr)
```

```
        if err != nil {
```

```
            panic(err)
```

```
        }
```

```
        body := &bytes.Buffer{}
```

```
// body1 := &bytes.Buffer{}
```

```
        fmt.Println("addr ", addr)
```

```
        fmt.Println("rsp main ", rsp)
```

```
        temp := strings.Split(addr, "/")
```

```
        name := temp[len(temp) - 1]
```

```

    leng, err := io.Copy(body, rsp.Body)

    fmt.Println(leng , " Bytes")

    if err != nil {

        panic(err)

    }

    fmt.Printf("%T \n", body)

    e := ioutil.WriteFile(name, body.Bytes(), 0644)

    if e != nil{

        panic(e)

    }

    //fmt.Printf("%v", body)

    fmt.Println("File Transferred")

    elapsed := time.Since(start)

    utils.Infof("%s", elapsed)

    wg.Done()

}{addr)

}

wg.Wait()

}

```

- Run the following commands at the client side:

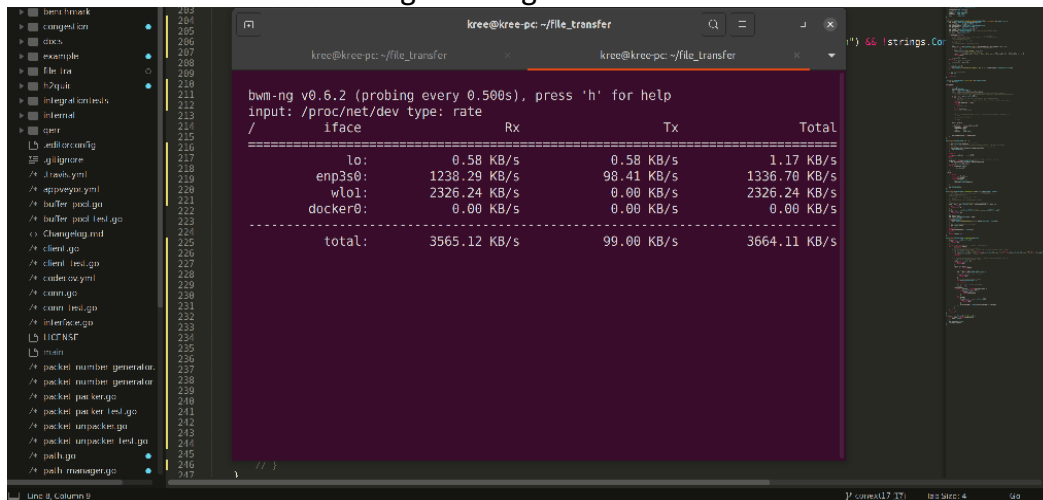
go build client.go

./client -m https://192.168.0.198:6121/a.tar.gz

- The command is in the following format:

`./client -m https://server-ip/file-name`

- Below is a screenshot bwm-ng showing file transfer over two interfaces



- Below is the terminal output at the client indicating path creation and file transfer.

```
1{fd01::3392:258:c4c:6950 45481 }{fd01::1834:8e4f:cdd8:9048 44385 }{192.168.0.103 35703 } {fd01::933d:a3c0:60d0:b9bf 37365 } {fd01::a23b:5154:235b:7b5f 37795 } {192.168.0.179 53327 } {fd01::3392:258:c4c:6950 45481 } {fd01::1834:8e4f:cdd8:9048 44385 }{192.168.0.103 35703 }

Started Path Creation

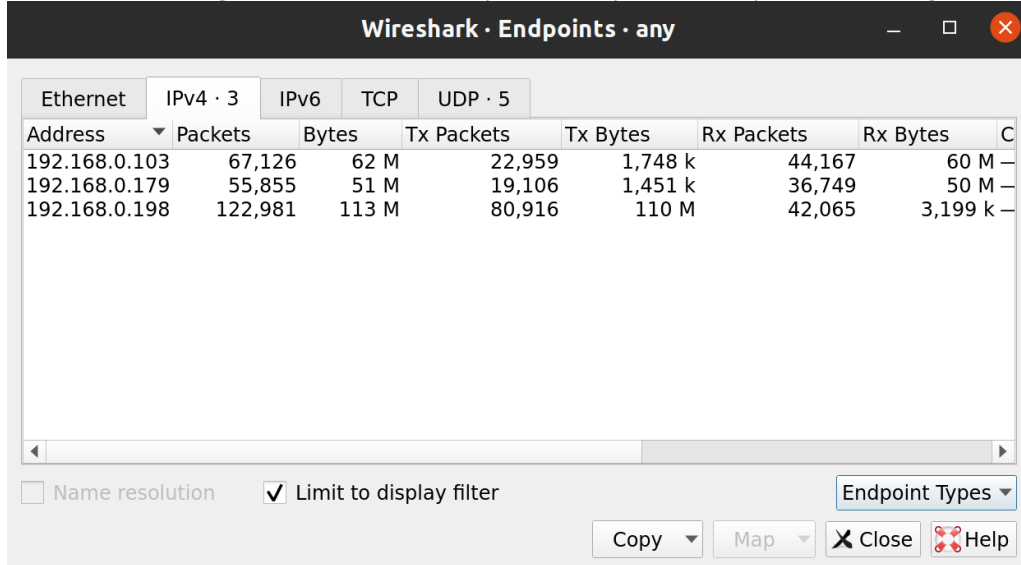
{fd01::933d:a3c0:60d0:b9bf 37365 }{fd01::a23b:5154:235b:7b5f 37795 }{192.168.0.179 53327 }

Started Path Creation

{fd01::3392:258:c4c:6950 45481 }{fd01::1834:8e4f:cdd8:9048 44385 }3addr https://192.168.0.198:6121/a.tar.gz
rsp main &{200 OK 200 HTTP/2.0 2 0 map[Accept-Ranges:[bytes] Content-Length:[104247844] Last-Modified:[Wed, 07 Apr 2021 05:35:20 GMT] Content-Type:[application/gzip]] 0xc4201d83c0 104247844 [] false false map[] 0xc42012e000 <nil>}
2021/05/09 00:15:22 Info for stream 5 of f6f5b4aa05939e3b
2021/05/09 00:15:22 Path 0: sent 6 retrans 0 lost 0; rcv 9
2021/05/09 00:15:22 Path 1: sent 20388 retrans 5 lost 0; rcv 40209
2021/05/09 00:15:22 Path 3: sent 21038 retrans 21 lost 3; rcv 40334
104247844 Bytes
*bytes.Buffer
hua...
2021/05/09 00:15:22 32.231995682s
```

- Path 0 is used for connection establishment. Path 1 and Path 3 are used for file transfer.

- Below is an image of wireshark endpoints of packets captured during file transfer:



The image shows the 'Wireshark · Endpoints · any' window. It displays a table of network endpoints with columns for Address, Packets, Bytes, Tx Packets, Tx Bytes, Rx Packets, and Rx Bytes. The data is as follows:

Address	Packets	Bytes	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes
192.168.0.103	67,126	62 M	22,959	1,748 k	44,167	60 M
192.168.0.179	55,855	51 M	19,106	1,451 k	36,749	50 M
192.168.0.198	122,981	113 M	80,916	110 M	42,065	3,199 k

At the bottom of the window, there are checkboxes for 'Name resolution' (unchecked) and 'Limit to display filter' (checked). There is also a dropdown menu for 'Endpoint Types' and buttons for 'Copy', 'Map', 'Close', and 'Help'.

- 192.168.0.103 is the ip address of LAN interface, inet 192.168.0.179 is the ip address of WiFi interface and inet 192.168.0.198 is the server ip address.
- Sum of the bytes sent on both the interfaces is equal to the number of bytes at the server. This validates the file transfer over multipath quic.

Video Streaming using MPQUIC using http requests

Requirements:

- A wifi router with at least 2 LAN connections.
- 2 Computers with MPQUIC installed. (Follow the above instructions)
- A LAN cable.
- Install the GoCV package.

Installing GoCV:

- Enter the following command to install **GoCv** in your system:

```
go get -u -d gocv.io/x/gocv
```

Aim: To stream video from one computer to the other using MPQUIC

Setup

- Connect both the computers to the router over WiFi using ssid and password.
- Here one computer acts as a client and the other as the server.
- We would like to have multiple paths at the client side, so connect the client computer to the router using a LAN cable.

Code Setup(Client side):

- Create a folder client_stream in the following directory:

```
~/go/src/github.com/lucas-clemente/quic-go/
```

- In client_stream, create a file client_stream.go and copy the following code:
- The client side code is as follows:


```
package main
```

```
import (
```

```
    "bytes"
```

```
    "crypto/tls"
```

```
    "flag"
```

```
    "io"
```

```
    "log"
```

```
    "net/http"
```

```
    "os"
```

```
    "sync"
```

```
    "time"
```

```
    "fmt"
```

```
    quic "github.com/lucas-clemente/quic-go"
```

```
    "github.com/lucas-clemente/quic-go/h2quic"
```

```
    "github.com/lucas-clemente/quic-go/internal/utils"
```

```
    "github.com/hybridgroup/mjpeg"
```

```
    "gocv.io/x/gocv"
```

```
)
```

```
var (
```

```
    err    error
```

```
    stream *mjpeg.Stream
```

)

```
func main() {  
    verbose := flag.Bool("v", false, "verbose")  
    multipath := flag.Bool("m", false, "multipath")  
    output := flag.String("o", "", "logging output")  
    cache := flag.Bool("c", false, "cache handshake information")  
    flag.Parse()  
    urls := flag.Args()  
    fmt.Println("urls", urls)  
  
    if *verbose {  
        utils.SetLogLevel(utils.LogLevelDebug)  
    } else {  
        utils.SetLogLevel(utils.LogLevelInfo)  
    }  
  
    if *output != "" {  
        logfile, err := os.Create(*output)  
        if err != nil {  
            panic(err)  
        }  
        defer logfile.Close()  
    }
```

```
    log.SetOutput(logfile)
}

quicConfig := &quic.Config{
    CreatePaths: *multipath,
    CacheHandshake: *cache,
}

hclient := &http.Client{
    Transport: &h2quic.RoundTripper{QuicConfig: quicConfig,
    TLSClientConfig: &tls.Config{InsecureSkipVerify: true}},
}

var wg sync.WaitGroup
wg.Add(len(urls))
for _, addr := range urls {
    utils.Infof("GET %s", addr)
    go func(addr string) {
        start := time.Now()
        rsp, err := hclient.Get(addr)
        if err != nil {
            panic(err)
        }
    }(addr)
}
```

```

    }

    body := &bytes.Buffer{}
    // body1 := &bytes.{ }
    fmt.Println("addr ", addr)
    fmt.Println("rsp main ", rsp)

    leng, err := io.Copy(body, rsp.Body)
    fmt.Println(leng , " Bytes")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T \n", body.Bytes())
    fmt.Println("hua...")
    elapsed := time.Since(start)
    utils.Infof("%s", elapsed)
    wg.Done()
    stream := mjpeg.NewStream()

    // start capturing
    go mjpegCapture()
}(addr)

```

```

    }

    wg.Wait()
}

// Video capture function using Webcam

func mjpegCapture() {
    img := gocv.NewMat()
    defer img.Close()
    for {
        if ok := gocv.NewMatFromBytes(480,640,1,body); !ok {
            fmt.Printf("Device closed: %v\n")
            return
        }
        if img.Empty() {
            continue
        }
        buf, _ := gocv.IMEncode(".jpg", img)
        stream.UpdateJPEG(buf)
    }
}

```

Code Setup(Server side):

- Create a folder `server_stream` in the following directory:

`~/go/src/github.com/lucas-clemente/quic-go/`

- In server_stream, create a file server_stream.go and copy the following code:
- The server side code is as follows:

```
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "path"
    "runtime"
    "strings"
    "sync"

    _ "net/http/pprof"

    "github.com/lucas-clemente/quic-go/h2quic"
    "github.com/hybridgroup/mjpeg"
    "gocv.io/x/gocv"
)

type binds []string

func (b binds) String() string {
    return strings.Join(b, ",")
}

func (b *binds) Set(v string) error {
    *b = strings.Split(v, ",")
    return nil
}

type Size interface {
    Size() int64
}
```

```

}

var (
    deviceID int
    err      error
    webcam   *gocv.VideoCapture
    imageStream *mjpeg.Stream
)

func init() {

    http.HandleFunc("/demo/echo", func(w http.ResponseWriter, r
    *http.Request) {
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
            fmt.Printf("error reading body while handling /echo:
%s\n", err.Error())
        }
        w.Write(body)
    })
}

func getBuildDir() string {
    _, filename, _, ok := runtime.Caller(0)
    if !ok {
        panic("Failed to get current frame")
    }

    return path.Dir(filename)
}

func main() {

    deviceID := 0
    host := "0.0.0.0:6121"

    webcam, err = gocv.OpenVideoCapture(deviceID)
    if err != nil {
        fmt.Printf("Error opening capture device: %v\n", deviceID)
    }
}

```

```

    return
}
defer webcam.Close()

imageStream = mjpeg.NewStream()

go mjpegCapture()

fmt.Println("Capturing. Point your browser to " + host)

go func() {
    log.Println(http.ListenAndServe("0.0.0.0:6060", nil))
}()

bs := binds{}
flag.Var(&bs, "bind", "bind to")
certPath := flag.String("certpath", getBuildDir(), "certificate
directory")

flag.Parse()

certFile := *certPath + "/fullchain.pem"
keyFile := *certPath + "/privkey.pem"

if len(bs) == 0 {
    bs = binds{"0.0.0.0:6121"}
}

var wg sync.WaitGroup
wg.Add(len(bs))
for _, b := range bs {
    bCap := b
    go func() {
        var err error

```



```

        err, server := h2quic.ListenAndServeQUIC(bCap, certFile,
keyFile, imageStream)

        if err != nil {
            fmt.Println(err)
        }
        wg.Done()
    }()
}
wg.Wait()
}

func mjpegCapture() {
    img := gocv.NewMat()
    defer img.Close()

    for {
        if ok := webcam.Read(&img); !ok {
            fmt.Printf("Device closed: %v\n", deviceID)
            return
        }
        if img.Empty() {
            continue
        }

        buf, _ := gocv.IMEncode(".jpg", img)
        imageStream.UpdateJPEG(buf)
    }
}

```

- The above runs a server at "localhost:6121" which is supposed to stream video from the client
- The stream from the client is captured and is recognised as mjpeg at the server but the packets are not streamed onto the http server, as there are some irregularities in the packet transmission over the multipath.

Video Streaming using MPQUIC sessions

Requirements:

- A wifi router with at least 2 LAN connections.
- 2 Computers with MPQUIC installed. (Follow the above instructions)
- A LAN cable.

Code Setup(Client side):

- Create a folder client_stream in the following directory:
~/go/src/github.com/lucas-clemente/quic-go/
- In client_stream, create a file client_stream.go and copy the following code:
- The client side code is as follows:

```
package main
```

```
import (  
    "crypto/tls"  
    "fmt"  
    "time"  
    "encoding/binary"  
  
    utils "./utils"  
    config "./config"  
    quic "github.com/lucas-clemente/quic-go"
```

```
    "gocv.io/x/gocv"
)

const addr = "192.168.0.103:"+config.PORT
var pl = fmt.Println

func main() {

    quicConfig := &quic.Config{
        CreatePaths: true,
    }

    pl("Trying to connect to: ", addr)
    sess, err := quic.DialAddr(addr,
&tls.Config{InsecureSkipVerify: true}, quicConfig)
    utils.HandleError(err)

    stream, err := sess.OpenStream()
    utils.HandleError(err)

    pl("Connected to server\n")
    pl("Starting Video streaming\n")
}
```

```

defer stream.Close()

webcam, _ := gocv.VideoCaptureDevice(0)

img := gocv.NewMat()

start := time.Now()

webcam.Read(&img)


var dims = make([]byte, 4)
var bs = make([]byte, 2)

binary.LittleEndian.PutUint16(bs, uint16(img.Rows()))
copy(dims[0:], bs)

binary.LittleEndian.PutUint16(bs, uint16(img.Cols()))
copy(dims[2:], bs)

stream.Write(dims)


var count = 1

for ;count<=config.MAX_FRAMES;count++){

    webcam.Read(&img)

    var b = img.ToBytes()

    for ind:=0;ind<len(b);{

        var end = ind+config.BUFFER_SIZE

        if end>len(b){

```

```

        end = len(b)
    }
    stream.Write(b[ind:end])
    ind = end
}

}

stream.Write([]byte{0,0,0,0})
webcam.Close()

elapsed := time.Since(start)
pl("\nEnded video streaming at: ", elapsed, "Frames Received
from server: ", count)

stream.Close()
stream.Close()

}

```

Code Setup(Server side):

- Create a folder server_stream in the following directory:
 ~/go/src/github.com/lucas-clemente/quic-go/

- In server_stream, create a file server_stream.go and copy the following code:
- The server side code is as follows:

```
package main
```

```
import (
```

```
    "fmt"
```

```
    "time"
```

```
    utils "./utils"
```

```
    config "./config"
```

```
    quic "github.com/lucas-clemente/quic-go"
```

```
    "gocv.io/x/gocv"
```

```
)
```

```
const addr = "0.0.0.0:" + config.PORT
```

```
var pl = fmt.Println
```

```
var p = fmt.Print
```

```
func main() {
```

```
    quicConfig := &quic.Config{
```

```
    CreatePaths: true,  
}
```

```
pl("Attaching to: ", addr)  
  
listener, err := quic.ListenAddr(addr,  
utils.GenerateTLSConfig(), quicConfig)  
utils.HandleError(err)
```

```
pl("Server listening...")
```

```
sess, err := listener.Accept()  
utils.HandleError(err)  
  
stream, err := sess.AcceptStream()  
utils.HandleError(err)
```

```
pl("Broadcasting incoming video stream...")  
defer stream.Close()
```

```
time.Sleep(10*time.Millisecond)  
  
start := time.Now()
```

```
buffer := make([]byte, config.BUFFER_SIZE)
```

```

// var frame gocv.Mat

var rows = -1

var cols = -1

window := gocv.NewWindow("Output")

var dims = make([]byte, 4)
stream.Read(dims)
rows = int(dims[1]) << 8 + int(dims[0])
cols = int(dims[3]) << 8 + int(dims[2])
pl("Video Dimensions : ", rows, " x ", cols)
var data = make([]byte, 3*rows*cols)
var dataind = 0

var count = 0
for ;count<config.MAX_FRAMES;{
    var limit = config.BUFFER_SIZE
    if limit+dataind >= len(data){
        limit = len(data)-dataind
        var temp = make([]byte, limit)
        stream.Read(temp)
        copy(data[dataind:],temp)
    }
}

```



```

        count++

        dataind = 0

        img,err := gocv.IMDecode(data,1)

        utils.HandleError(err)

        window.IMShow(img)

        window.WaitKey(1)
    } else{

        stream.Read(buffer)

        copy(data[dataind:],buffer)

        dataind = dataind+limit

    }
}

elapsed := time.Since(start)

pl("\nEnding video transmission, Duration: ", elapsed, "
Frames Captured ", count)

stream.Close()

stream.Close()

pl("\n\nThank you!")
}

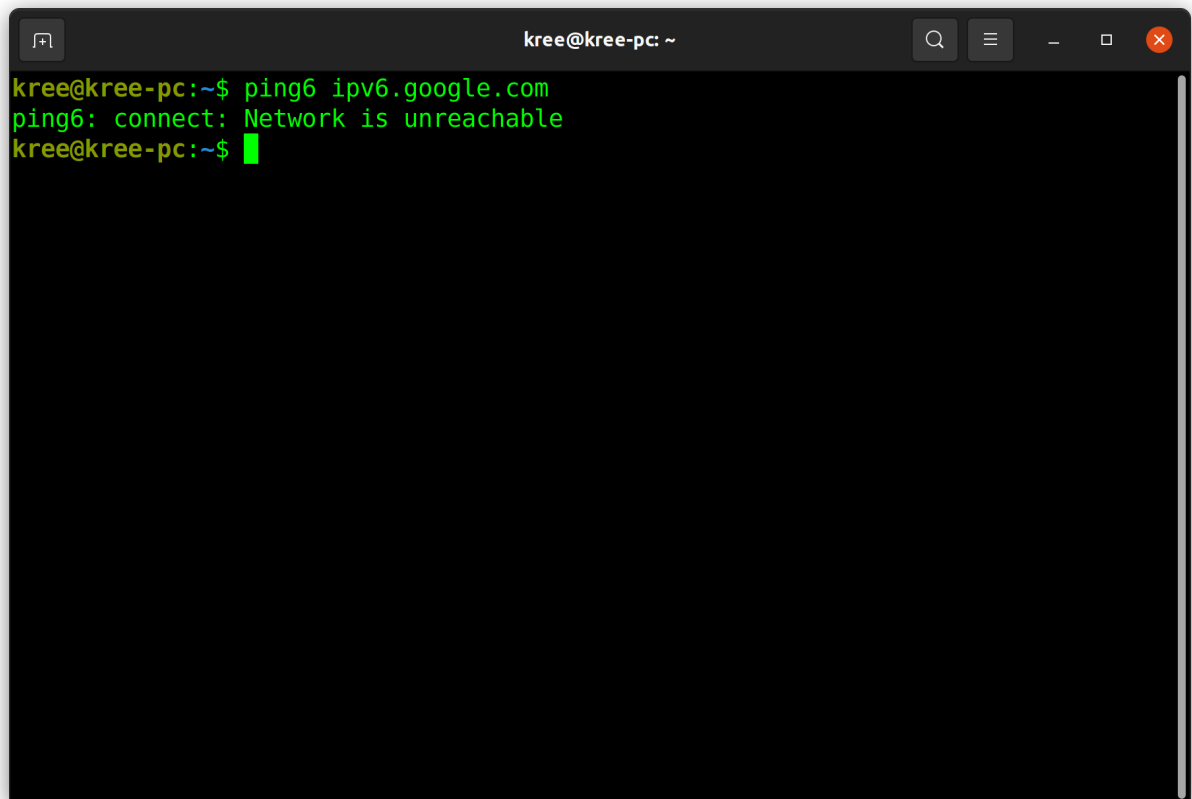
```

- This approach gave better results. The video streamed on the client side but resulted in discolored images.
- This was because of how the image data is being sent by the scheduler.
- This can be rectified by using different scheduling algorithms for sending image data.

File transfer using MPQUIC with IPv6

- The file transfer uses a code IPv4 http server.
- IPv4 server has been changed to listen on localhost of IPv6 of the server that is `:::1:6161`
- The connection was established between the server and the client but there was no packets received from the server.
- This could be because IPv6 forwarding was not enabled in `/etc/sysctl.conf` file.
- Made the following change in `/etc/sysctl.conf` :

```
net.ipv6.ip_forward=1
```
- The above change could not give the expected results.
- This could be because the network could be blocking all IPv6 packets since the

A terminal window titled 'kree@kree-pc: ~' with standard window controls. The terminal shows a command 'ping6 ipv6.google.com' being executed, followed by the output 'ping6: connect: Network is unreachable'. The prompt 'kree@kree-pc:~\$' is shown again with a green cursor.

```
kree@kree-pc:~$ ping6 ipv6.google.com
ping6: connect: Network is unreachable
kree@kree-pc:~$
```

following command to ping IPv6 packets to google resulted ***ping6: connect:***

Network is unreachable

- Tried making a few other changes in the linux system code but could not transfer files using IPv6.

Observations

- There are two schedulers available for MPQUIC. The two schedulers are Low Latency scheduler and Round Robin.
- MPQUIC uses Round Robin as default.
- The connection manager is `pconn_manager.go`, it probes for connected interfaces on the client side.
- The path manager is `path_manager.go`, it uses the captured interfaces and creates multiple paths at the client side to receive from the server.
- There are also two congestion control algorithms for MPQUIC. OLIA is default for MPQUIC and there is also CUBIC which is a similar implementation of MPTCP congestion control algorithm.
- When multiple paths are not available, mpquic uses CUBIC for congestion. If there are multiple paths, it uses OLIA for congestion. This is because the code assumes that the absence of multiple paths is congestion.
- The above observation was made by disabling one of the paths during run backtracking the function that is called during congestion.
- Restricting the bandwidth using `netem` also does not affect path selection.
- The video streamed on the client side but resulted in discolored images.
- This was because of how the image data is being sent by the scheduler.
- This can be rectified by using different scheduling algorithms for sending image data.

Future work

To implement video streaming using different streaming algorithms.

References

1. Quentin De Coninck and Olivier Bonaventure. 2017. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 160–166.
DOI:<https://doi.org/10.1145/3143361.3143370>
2. T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe and R. Steinmetz, "Multipath QUIC: A Deployable Multipath Transport Protocol," *2018 IEEE International Conference on Communications (ICC)*, Kansas City, MO, USA, 2018, pp. 1-7, doi: 10.1109/ICC.2018.8422951.
3. Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 183–196.
DOI:<https://doi.org/10.1145/3098822.3098842>
4. H. Wu, Ö. Alay, A. Brunstrom, S. Ferlin and G. Caso, "Peekaboo: Learning-Based Multipath Scheduling for Dynamic Heterogeneous Environments," in *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2295-2310, Oct. 2020, doi: 10.1109/JSAC.2020.3000365.
5. Alexander Rabitsch, Per Hurtig, and Anna Brunstrom. 2018. A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths: Paper # XXX, XXX pages. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ'18)*. Association for Computing Machinery, New York, NY, USA, 29–35.
DOI:<https://doi.org/10.1145/3284850.3284855>