ping を用いて 2 ホストまで経由する経路を探索する
ファイル協調ダウンロードシステム

(A 班共通レポート)

メンバー名

50916036 新村嘉基　リーダー

竹澤大樹　副リーダー

SURINA　メンバー

1. 設計したシステムの戦略

1.1 設計したシステムの基本的な戦略

今回作成したファイル協調ダウンロードシステムは大きく分けて、パケットロスのない経路の探索、SIZE コマンド、最適経路の選択、GET コマンド、REP コマンドの 5 つからなる。以下それぞれについて説明していく。

1.1.1 パケットロスのない経路の探索

AWS 上でファイルダウンロードするプログラムを実行していくうち、TCP 通信ではパケットロスをする経路を除外できるかどうかで全体の通信品質に大きな違いがでることに気付いた。そのため、本システムではパケットロスをする経路をできるだけ選択しないようにした。

○ 探索経路

経路は転送管理サーバを 2 ホストまで経由できるものとする。

(7 ホストの場合は 1(サーバと直接)+5(1 ホスト経由)+20(2 ホスト経由) = 26 通り。)

転送管理サーバを	1つ挟む	2つ挟む
	(マシンの数-2)+1	(マシンの数-2)×(マシンの数-3) +(マシンの数-2)+1
マシンが４台	3	5
マシンが７台	6	26

○ ping のパラメータ

探索経路がパケットロスをするかどうかは python 標準モジュールの subprocess を用いてプログラム上で ping コマンドを実行することで調べている。探索において ping の送信サイズは 60kByte、送信回数 20 回、送信間隔 0.2 秒にしている。これはパケットロス率を調べるという目的とは別に、高頻度で大きいサイズのパケットを送ることで経路を輻輳させ、輻輳に弱い経路を除外できるのではないかと考えたためである。実際にこの条件で実行してみたところ、いくつか輻輳に弱いと考えられる経路を除外することができた。

○ パケットロス調査

2 ホストまでの経由する経路のうち、その経路内にパケットロスのないものを選択する。パケットロスの調査を行う際、ping を行うことに加えて Route パケットを各ホストへ送信することで最終的に指定した経路での ping による RTT の累積などもクライアント側に伝えている。

○ RouteTable の作成

クライアント側で指定した経路の情報とその経路での ping による RTT の累積を格納した二次元配列である RouteTable を作成する。その後、作成した RouteTable を ping による RTT の総和でソートする。

1.1.2 SIZE コマンド

サーバに対して SIZE コマンドを行い、要求するファイルサイズの情報を得る。このとき使用する経路は ping による RTT の累積が最も小さいものである。これは SIZE コマンド用のパケットを送信することで実装している

1.1.3 最適経路の選択

適切な複数の経路を使用しファイルのデータを分割してダウンロードすれば競技時間は短縮できる。しかし、経路に送るデータ量を誤れば輻輳が発生し競技時間が大幅に長くなってしまう。このような問題を避けるために最適経路の選択を行う。具体的な経路選択方法は以下の通り。

基本的にはファイルサイズによって以下のように決定する。

1MB 未満の場合は 4 経路選択

1MB 以上 3MB 未満の場合は 3 経路選択

3MB 以上 4MB 未満の場合は 2 経路選択

4MB 以上の場合は 1 経路選択

次に、あまりにも RTT が他の経路と比べて大きいものを除外する処理を行う。

例えば、ping による RTT の累積が最も小さい経路と比べて二番目の経路は帯域幅があまりにも小さい場合は経路一つでファイルをダウンロードする。そうでない場合は二つ目の経路も使用するなどである。

これは昨年のＡＷＳ上の環境を基に今年の環境を考えた。ファイルを分割し十分にダウンロードが速く行えるのは、4 本経路を使い分割してダウンロードする場合までだということで 4 経路まで経路の選択は行っている。また一本目の経路が十分に帯域が太くなっていると予想したのは転送管理サーバを 2 つ挟んだことによって最適経路を考えた場合、昨年の環境なら十分に帯域が太かったからである。

1.1.4 GET コマンド

まず、「最適経路の選択」で選択した各経路でダウンロードするファイルサイズを決定する。ファイルを分割するサイズは SIZE コマンドにより得られた全体のファイルサイズを各経路における ping による RTT の累積の逆数の比で分けることにより決定する。次に、GET_PARTIAL コマンドを行いそれぞれの経路を用いて分割ファイルをダウンロードするこれは GET コマンド用パケットを送信することで実装している

1.1.5 REP コマンド

サーバに対して REP コマンドを行う。このとき使用する経路は ping から得られた RTT の総和が最も小さい経路である。REP コマンド用パケットを送信することで実装している

2 システムの実装

2.1 プログラム一覧・機能概要

○ クライアント側(client_p2s.py)

機能：Route 用パケットの作成・送信・受け取り、各種コマンド用パケットの作成・送信･受け取り､分割ファイルサイズ決定､最適経路選択､指定したホストへの ping

○ 転送管理側(mid_p2s.py)

機能：各種パケットの送信・受け取り・書き換え、指定したホストへの ping

2.2 プログラムとソースコードの関係

- グローバル変数の説明

グローバル変数として設定した変数とその説明は以下コメントの通り。

```
# -----クライアント設定------------
my_name = os.uname()[1]  # クライアントのホスト名あるいはIPアドレスを表す文字列
my_port = 53602 # クライアントのポート
my_port_route = 53625 # クライアントのポート(念のため)
my_port_size = 53624 # クライアントのポート(size)
my_port_get = 53623 # クライアントのポート(get)
my_port_rep = 53622 # クライアントのポート(rep)

# ----------サーバ(転送管理サーバ)設定--------------
server_name = sys.argv[1] # サーバのホスト名
server_port =  60623 # サーバのポート
server_file_name = sys.argv[2] # サーバ側にあるファイル名
mid_name = '' # 中間管理サーバの名前
mid_port = 53608 # 中管理サーバのポート

# ----- Route用設定--------------
RouteTable = [] # 調べた経路を保存するリスト
# address = ["pbl1","pbl2","pbl3","pbl4"] # ローカル環境のアドレス
address = ["pbl1a","pbl2a","pbl3a","pbl4a", "pbl5a","pbl6a","pbl7a"]
ad_first = [] # 送信する1つめのホストはpingによって絞る

# -------コマンド用設定------------
key = '' # トークンキー
token_str = sys.argv[3] # トークン文字列
sdata_num = 0 # ファイル受け取りを正当な順番でするために必要な変数
rec_file_name = 'received_data.dat' # 受け取ったデータを書き込むファイル
BUFSIZE = 1024 # 受け取る最大のファイルサイズ
```

次に、client_p2s.py と mid_p2s.py のプログラム説明をする。

○ client_p2s.py

‣ プログラムの関数等の説明

・rec_res 関数

引数：ソケット

```python
# \nまでの文字列の受け取り(\nを含まない)
def rec_res(soc):
    # 応答コードの受け取り
    recv_bytearray = bytearray() # 応答コードのバイト列を受け取る配列
    while True:
        b = soc.recv(1)[0]
        if(bytes([b]) == b'\n'):
            rec_str = recv_bytearray.decode()
            print(rec_str)
            break
        recv_bytearray.append(b)
    # print('received')

    return rec_str
```

コネクションしているホストから’¥n’まで文字列を受け取る。

・SIZE 関数

引数；ファイル名

```python
# SIZE(文字列生成)
def SIZE(file_name):
    # 要求
    msg = f'SIZE {file_name}' # 要求メッセージ
    # soc.send(msg.encode())
    # print('request SIZE')
    return msg
```

SIZE 要求用の文字列を作成し、返り値として返す

・GET_part 関数

引数：ファイル名、token 文字列，分割ファイルサイズの先頭，分割ファイルサイズの末端

```python
# GET(PARTIAL)
def GET_part(file_name,token_str,sB, eB):
    # 要求
    key = pbl2.genkey(token_str) # keyの作成
    msg = f'GET {file_name} {key} PARTIAL {sB} {eB}' # 要求メッセージ

    # print('request GET PARTIAL')
    return msg
```

GET_PARTIAL 要求用の文字列を生成し、返り値として返す。

・REP 関数

引数：ファイル名

```
88  # REP(文字列生成)
89  def REP(file_name):
90      key = pbl2.genkey(token_str) # keyの作成
91      repkey_out = pbl2.repkey(key, rec_file_name) # repkeyの作成
92      msg = f'REP {file_name} {repkey_out}' # 要求メッセージ
93      # print(msg)
94      # print('request REP')
95      return msg
```

REP 要求用の文字列を生成し、返り値として返す。

・receive_server_file 関数

引数：ソケット，何番目のデータ分割かを表す変数

```
98  def receive_server_file(soc, s_data_num):
99      # 書き込み用ファイルをオープンして処理
100     #   ファイル絡みの例外処理とクローズの処理は書く必要がありません
101     if s_data_num==0: #新規ファイル作成
102         com='wb'
103     elif s_data_num>=1:#既存ファイルに追記
104         com='ab'
105     with open(rec_file_name, com) as f:
106         while True:
107             data = soc.recv(BUFSIZE)    # BUFSIZEバイトずつ受信
108             if len(data) <= 0:  # 受信したデータがゼロなら、相手からの送信は全て終了
109                 break
110             f.write(data)  # 受け取ったデータをファイルに書き込む
```

コネクションされたホストから送られてきた文字列をファイルに書き込む関数。

1 番目のデータ分割であればファイルを新規作成、それ以外であれば追記で書き込む。

・Ping 関数

引数：ping を行う宛先のホスト名

パラメータ設定部分

```
171 # ping
172 def Ping(ad):
173     # 正規表現'%'が後ろにつく[0,100]の数字を検索するための正規表現オブジェクトを生成
174     # regex = re.compile(r'\s[0-100](?=%)')
175     regex = re.compile(r'\d{1,3}(?=%)')
176
177     ping = subprocess.run(
178         ["ping", "-c", "20","-i", "0.2","-s","65507","-q", ad],
179         stdout=subprocess.PIPE,     # 標準出力は判断のため保存
180         stderr=subprocess.DEVNULL # 標準エラーは捨てる
181     )
```

指定したホストに対して ping を行う。ping は 65507B を 0.2 秒間隔で 20 回送信するようパラメータを設定する。

出力の受け取り・パケットロス率と RTT の抜出

```
    # pingの出力
    output = ping.stdout.decode("cp932")
    # print(output)
    # outputからパケロスの数値を抽出する
    packet_loss = regex.search(output)

    # pingコマンドが成功->パケットロス率を返す,失敗->パケットロス率を100%とする
    if packet_loss == None:
        return_p_loss = 100
    else:
        return_p_loss = float(packet_loss.group())
     # outputからrttを抽出する
     if(return_p_loss != 100):
         # outputからrttの平均を抽出する
         i = output.find('rtt')
         rtt_info = output[i:]
         rtt_info = re.split('[/ ]', rtt_info)
         # print(rtt_info)
         rtt = float(rtt_info[7])
     else:
         rtt = 1000000

     print()
     print('ping to',ad)
     print('packetloss:',return_p_loss,'rtt',rtt)
     print()

     return ad, return_p_loss, rtt
```

実行した ping の出力文字列からパケットロス率と RTT の抜出を行う。
返り値として指定した宛先，パケットロス率, RTT を返す。

・Ping_ALLHost 関数

引数：なし

```
# すべてのホストにPing(並列処理)
def Ping_AllHost():
    with ThreadPoolExecutor(max_workers = 4) as executor:
        # tupleで受けとり
        route_info = list(executor.map(Ping, (ad for ad in address)))

    return route_info
```

Ping 関数をマルチスレッドを用いて並列で全ホストに対して実行するための関数。
Ping 関数の返り値は route_info 変数でタプルとして受け取り、これを関数の返り値として返す。

・select_ad_first 関数

引数：route_info 変数(Ping の返り値をタプルで受け取った変数), 許容するパケットロス率

```
229  # ad_firstを選択
230  def select_ad_first(route_info, p_loss_lim):
231      for ad, p_loss, rtt in route_info:
232          if(p_loss <= p_loss_lim):
233              if(ad == server_name):
234                  RouteTable.append([rtt, my_name, 'none', 'none', \
235                                     server_name])
236              ad_first.append((ad, rtt)) # tupleでappendする
237
```

クライアントから Route パケットを送る 1 経由目のホストを決定する。サーバ以外のホストに対しての通信のパケットロス率が許与範囲内だった場合は ad_first 変数にそのホスト名を格納。また、サーバと直接通信のパケットロス率が許容範囲内だった場合は RouteTable に ping から得られた RTT と経路情報を格納。

・fix_route_packet 関数

引数：Route パケットの配列

```
112  # route/パケットを適切な型に変換
113  def fix_route_packet(pack):
114      if(pack[4] == 'False'):
115          pack[4] = False
116      elif(pack[4] == 'True'):
117          pack[4] = True
118
119      pack[5] = int(pack[5])
120      pack[6] = int(pack[6])
121      pack[9] = int(pack[9])
122      pack[10] = float(pack[10])
123
124      return pack
125
```

Route パケットの配列の各要素を文字列から適切な型に変換する。

・exchange_route_packet 関数

引数：一番目に経由するホスト名, 2 番目に経由するホスト名(ない場合は'none'), TTL,
　　指定した経路での RTT の累積

Route 用パケット

Route パケットの中身は

「クライアント名/経由する 1 ホスト目/経由する 2 ホスト目/サーバ名/到達性フラグ/TTL/経由回数/パケットの種類(Route)/送信用 or 応答用/Route 用のクライアントポート番号/指定した経路での ping による RTT の累積」

となっており。送信するパケットは文字列で各要素'¥n'区切りになっている

Route パケットの作成

```python
# ルーティングパケットの送受信
def exchange_Routepacket_ping(ad1, ad2, ttl, rtt):
    global my_port_route

    # -------Routeパケットの送信-------------
    client_socket = socket(AF_INET, SOCK_STREAM)
    client_socket.connect((mid_name, mid_port)) # 送信するホストとコネクション

    # パケット到達性フラグ
    flg_route = True
    # 経由回数
    relay_num = 1

    """
    Route用パケット
    クライアント名\経由する1ホスト目\経由する2ホスト目\
            サーバ\パケット到達性フラグ\TTL\経由回数\
            パケットの種類(Route)\送信用(req) or 応答用(rep)\
            Route用のクライアントのポート番号\指定した経路でのrttの累積
    """
    # パケットは / で区切る
    info_pack = f"{my_name}/{ad1}/{ad2}/{server_name}/{flg_route}/{ttl}/\
        {relay_num}/Route/req/{my_port_route}/{rtt}"
    info_pack += '\n'
    start_time = time.time()
```

TTL は 1 ホスト経由の場合は 1、2 ホスト経由の場合は 2 と設定することで転送管理側で 1 ホスト経由の経路か 2 ホスト経由の経路か判断している。また、relay_num(Route パケットが何ホスト経由したか表す変数)をはじめ 1 に設定している。

Route パケットの送受信

```python
    client_socket.send(info_pack.encode()) # データ配列の送信
    rep_info =  rec_res(client_socket)
    end_time = time.time()
    # /で区切ってリストに格納(各要素は文字列として認識される→適切な型への変換が必要)
    rep_info_pack = rep_info.split('/')
    # 受け取ったパケットを適切な型変換
    rep_info_pack = fix_route_packet(rep_info_pack)
    client_socket.close()

    # 2ホスト経由の場合
    # else:
    client_socket.close()
    return rep_info_pack, (end_time - start_time)
```

Route パケットの送受信を行う。受け取った返答用 Route パケットは 156 行目で各要素を配列に格納している。このとき、格納した要素はすべて文字列になっているので適切な要素の型に変更する必要がある。(158 行目)

・routing_1host 関数

引数：なし

```
239  # ホスト1つを経由する場合の経路を調べる
240  def routing_1host():
241      ttl = 1
242      # 経路はあらかじめクライアント側で決定しておく
243      # 自分とserver_nameを持つ転送管理サーバ以外へRoute用パケットを送信
244      for ad, rtt in ad_first:
245          if my_name != ad and server_name != ad:
246              future = tpe.submit(exchange_Routepacket_ping, ad, 'none', ttl, rtt)
247              futures.append(future)
```

1 ホスト経由の Route パケット送信タスクを実行キューに追加する。

1 ホスト目のホスト名を ad_first リストから用いることでパケットロスのある経路に Route パケットを送らないようにしている。

・routing_2host 関数

引数：なし

```
249  # ホスト2つを経由する場合の経路を調べる
250  def routing_2host():
251      # 経路はあらかじめクライアント側で決定しておく
252      # 自分とserver_nameを持つ転送管理サーバ以外へRoute用パケットを送信
253      ttl = 2
254      for ad1, rtt in ad_first:
255          if my_name != ad1 and server_name != ad1:
256              for ad2 in address:
257                  if my_name != ad2 and server_name != ad2 and ad1 != ad2:
258                      # Route用パケットのやり取り
259                      future = tpe.submit(exchange_Routepacket_ping, ad1, ad2, ttl, rtt)
260                      futures.append(future)
158
159      rep_info =  rec_res(client_socket)
160      end_time = time.time()
161      # /で区切ってリストに格納(各要素は文字列として認識される→適切な型への変換が必要)
162      rep_info_pack = rep_info.split('/')
163      # 受け取ったパケットを適切な型変換
164      rep_info_pack = fix_route_packet(rep_info_pack)
165      client_socket.close()
166
167      # 2ホスト経由の場合
168      # else:
169      client_socket.close()
170      return rep_info_pack, (end_time - start_time)
```

2 ホスト経由の Route パケット送信タスクを実行キューに追加する。

1 ホスト目のホスト名を ad_first リストから用いることでパケットロスのある経路に Route パケットを送らないようにしている。

・recv_Route_packet 関数

引数：タイムアウト時間

```
262  # tpeの実行(Routeパケットの受け取り)
263  def recv_Route_packet(TO_time):
264      global route_count
265      for future in futures:
266          try:
267              # Routeの関数を並列実行
268              rep_info_pack, time = future.result(timeout=TO_time)
269              if(rep_info_pack[4] == True):
270                  '''
271                  ルーティング結果
272                  Route = [指定した経路での累積rtt, クライアント名,
273                          経由する1ホスト目, 経由する2ホスト目, サーバ]
274                  '''
275                  time_table.append((rep_info_pack[:3],time, rep_info_pack[10]))
276                  Route = [rep_info_pack[10], rep_info_pack[0],rep_info_pack[1],\
277                          rep_info_pack[2], rep_info_pack[3]]
278                  # Routeテーブルに各経路でのルーティング結果を格納
279                  RouteTable.append(Route)
280              # print('time', time,'rtt', rep_info_pack[10])
281          except TimeoutError:
282              print('Timeout Erro')
283          except:
284              print(sys.exc_info())
```

実行キューに追加した Route パケット送信タスクを並列に実行し、その返り値を受け取る。また、ここで返ってきた Route パケットから RouteTable を作成している。RouteTable とは各経路ごとのルーティング結果(コメント 270~274 行目)を格納した 2 次元配列である。タイムアウト時間を設定しており、タイムアウト時間が経過した場合はタイムアウト例外処理をそれ以外の場合はそれ以外の例外処理を行う。

・load_data_size 関数

引数：SIZE 応答メッセージ

```
286  # SIZE応答からデータサイズを読み取り
287  def load_data_size(SIZE_msg):
288      msg_list = SIZE_msg.split() # 空白で分割
289      data_size = int(msg_list[2]) # size部分を取り出す
290      return data_size
```

SIZE 応答用メッセージからデータサイズを抜き出す。

・SIZE_cmd 関数

引数：RouteTable

[1] サーバと直接の通信する経路での SIZE コマンド

```
292  # SIZEパケットのやり取り(サーバ側へ最も速い経路をたどってSIZE要求をする)
293  def SIZE_cmd(RouteTable):
294
295      # サーバ側に対して直接SIZEコマンドを実行
296      if(RouteTable[0][2] == 'none'):
297          # -------SIZEコマンド要求の送信---------
298          client_socket = socket(AF_INET, SOCK_STREAM)
299          client_socket.connect((server_name, server_port))
300          size_msg = SIZE(server_file_name)
301          size_msg += '\n'
302          client_socket.send(size_msg.encode())
303
304          # -------SIZEコマンド応答の受け取り---------
305          SIZE_sentence = rec_res(client_socket)
306          # print('SIZE_sentence', SIZE_sentence)
307          if SIZE_sentence[0:2] == "NG":
308              print("check your filename and try again")
309              sys.exit( )
310          data_size = load_data_size(SIZE_sentence)
311          client_socket.close()
```

SIZE 用の要求メッセージ作成しサーバ側に送信する。その後 SIZE 用応答メッセージをサーバから受け取る。サーバと直接通信する経路であるかどうかは RouteTable の各行の要素番号 2 (1 ホスト目のホスト名)が'none'あるかどうかで判定している

[2] サーバと直接の通信する経路以外の経路での SIZE コマンド

SIZE 用パケットの作成

```
299      # 転送管理サーバを挟んだ経路でのサイズ要求
300      else:
301          # -------SIZEパケットの送信---------
302          client_socket = socket(AF_INET, SOCK_STREAM)
303          client_socket.connect((RouteTable[0][2], mid_port))
304          relay_num = 1
305          """
306          SIZE用パケット
307          SIZE_pack = [クライアント名, 経由する1ホスト目, 経由する2ホスト目,
308                       サーバ,コマンドの種類(SIZE), コマンド要求文字列,
309                       経由回数, パケットの種類(Com), 要求(req) or 応答(rep),
310                       size用のクライアントのポート番号]
311          """
312          SIZE_pack = f"{RouteTable[0][1]}/{RouteTable[0][2]}/{RouteTable[0][3]}/\
313              {RouteTable[0][4]}/SIZE/{SIZE(server_file_name)}/{relay_num}/Com/req/\
314              {my_port_size}"
315          SIZE_pack += '\n'
316
```

SIZE 用パケット

SIZE パケットの中身は

「クライアント名/経由する 1 ホスト目/経由する 2 ホスト目/サーバ名/コマンドの種類(SIZE)/コマンド要求文字列/経由回数/パケットの種類(Com)/送信用 or 応答用/SIZE 用のクライアントポート番号」となっており。送信するパケットは文字列で各要素'¥n'区切りになっている

SIZE 用パケットの送信、応答文字列の受け取り

```
331         client_socket.send(SIZE_pack.encode()) # データ配列の送信
332
333         # -------SIZEsentenceの受け取り---------
334         # クライアント側にはSIZEコマンドの応答(string)が送られてくる
335         SIZE_sentence = rec_res(client_socket)
336         if SIZE_sentence[0:2] == "NG":
337             print("check your filename and try again")
338             sys.exit( )
339         # print('SIZE_sentence', SIZE_sentence)
340         data_size = load_data_size(SIZE_sentence)
341         client_socket.close()
342         # パケットのサイズを返す
343     return data_size
```

SIZE 用パケットの送信をし、SIZE 応答文字列を受け取る。336 行目~338 行目では SIZE 応答が NG であった場合の例外処理をしている。また、340 行目で SIZE 応答文字列からデータサイズを抜き出し、これを返り値とする。

・GET_part_send 関数

引数：ソケット, RouteTable, token 文字列,サーバにあるファイル名,データ分割先頭バイト,
　　データ分割末端バイト

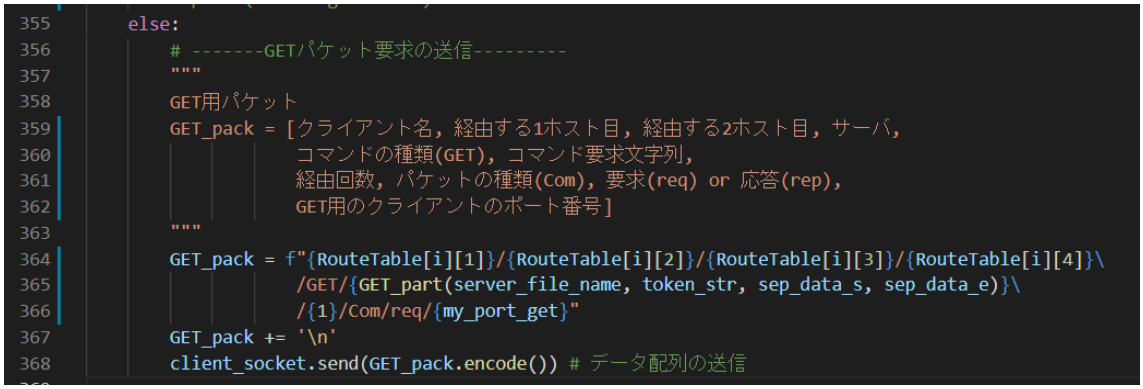
[1] サーバと直接の通信する経路での GET 要求

```
345 # GET_partialコマンド(送信)
346 def GET_part_send(client_socket, RouteTable, token_str, \
347     server_file_name, sep_data_s, sep_data_e, i):
348
349     if(RouteTable[i][2] == 'none'):
350         # -------GETコマンド要求の送信---------
351         get_msg = GET_part(server_file_name, token_str, sep_data_s, sep_data_e)
352         get_msg += '\n'
353         client_socket.send(get_msg.encode())
354         # print('sending server')
```

サーバへ GET 要求文字列を送信する。サーバと直接通信する経路であるかどうかは RouteTable の各行の要素番号 2 (経由する 1 ホスト目のホスト名)が'none'あるかどうかで判定している

[2] サーバと直接の通信する経路以外の経路での GET 要求

GET 用パケットの作成・送信



```
355     else:
356         # -------GETパケット要求の送信---------
357         """
358         GET用パケット
359         GET_pack = [クライアント名, 経由する1ホスト目, 経由する2ホスト目, サーバ,
360                     コマンドの種類(GET), コマンド要求文字列,
361                     経由回数, パケットの種類(Com), 要求(req) or 応答(rep),
362                     GET用のクライアントのポート番号]
363         """
364         GET_pack = f"{RouteTable[i][1]}/{RouteTable[i][2]}/{RouteTable[i][3]}/{RouteTable[i][4]}\
365                     /GET/{GET_part(server_file_name, token_str, sep_data_s, sep_data_e)}\
366                     /{1}/Com/req/{my_port_get}"
367         GET_pack += '\n'
368         client_socket.send(GET_pack.encode()) # データ配列の送信
```

GET 用パケット

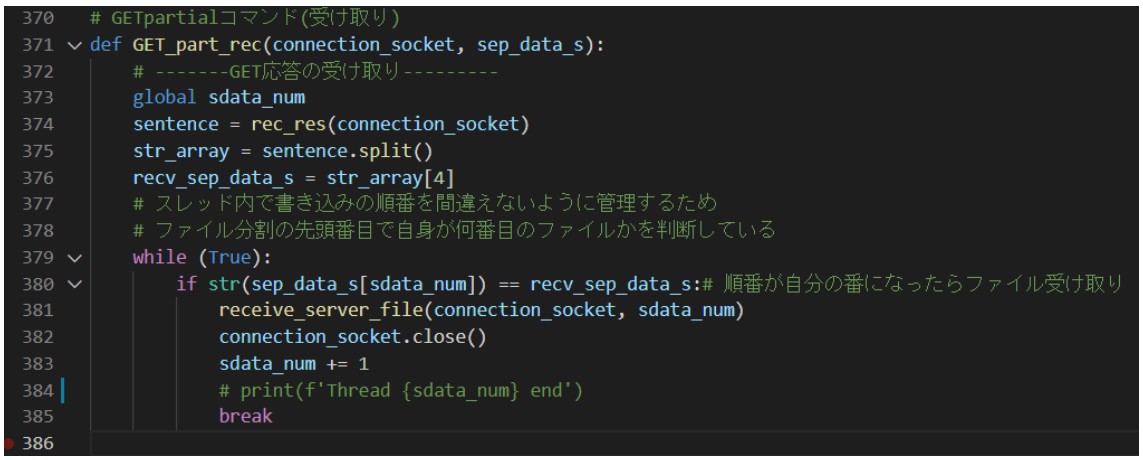
GET パケットの中身は

「クライアント名/経由する 1 ホスト目/経由する 2 ホスト目/サーバ名/コマンドの種類(GET)/コマンド要求文字列/経由回数/パケットの種類(Com)/送信用 or 応答用/GET 用のクライアントポート番号」となっており。送信するパケットは文字列で各要素'¥n'区切りになっている

・GET_part_rec 関数

引数：ソケット，各データ分割の先頭バイトを格納したリスト

GET 応答用文字列，分割したファイルの中身の受け取り。



```
370 # GETpartialコマンド(受け取り)
371 def GET_part_rec(connection_socket, sep_data_s):
372     # -------GET応答の受け取り---------
373     global sdata_num
374     sentence = rec_res(connection_socket)
375     str_array = sentence.split()
376     recv_sep_data_s = str_array[4]
377     # スレッド内で書き込みの順番を間違えないように管理するため
378     # ファイル分割の先頭番目で自身が何番目のファイルかを判断している
379     while (True):
380         if str(sep_data_s[sdata_num]) == recv_sep_data_s:# 順番が自分の番になったらファイル受け取り
381             receive_server_file(connection_socket, sdata_num)
382             connection_socket.close()
383             sdata_num += 1
384             # print(f'Thread {sdata_num} end')
385             break
386
```

分割したファイルの中身を順番通りに受け取るために分割データの先頭バイトを格納したリストとグローバル変数の sdata_num を用いている。初め sdata_num＝0 と設定しておき、送られてきた GET_PARTIAL 応答文字列の分割データの先頭バイト部分が sep_data_s[sdata_num]だった場合そのファイルの中身を受け取り sdata_num を 1 増やす。これを繰り返すことで分割したファイルの中身を順番通りに受け取っている。

・GET_cmd_part 関数

引数：RouteTable，トークン文字列，サーバにあるファイル名，データサイズ

データ分割

```
387  # GETコマンド
388  def GET_part_cmd(RouteTable, token_str, server_file_name, data_size):
389      # print()
390      # print('separate data')
391      sep_data_s=[] # 分けたデータの最初を入れる
392      sep_data_e=[] # 分けたデータの最後を入れる
393
394      # データ分割(ルーティングパケットの往復時間依存)
395      ratio_list = []
396      SumRatio = 0
397      # (1 / 計測時間)のリスト作成
398      for i in range(0, len(RouteTable)):
399          ratio_list.append(1. / RouteTable[i][0])
400          SumRatio += (1. / RouteTable[i][0])
401
402      for i in range(0,len(RouteTable)):
403          if i == 0:
404              separate_data_s=0
405          else:
406              separate_data_s=separate_data_e+1
407          separate_data_size = int(float(data_size)*((ratio_list[i]/ SumRatio)))
408          if(i == len(RouteTable)-1):
409              separate_data_e = data_size
410          else:
411              separate_data_e = separate_data_size + separate_data_s
412
413          # print(f'sep{i} data_size: {separate_data_e - separate_data_s}')
414
415          sep_data_s.append(separate_data_s)
416          sep_data_e.append(separate_data_e)
```

データファイルサイズを指定した経路の ping による RTT の累積の逆数の比で分割する。ここでは RouteTable の各行の要素番号 0(指定した経路の ping による RTT の累積)を用いて実装している。

sep_data_s リストに各分割におけるデータの先頭バイトを sep_data_e リストに各分割におけるデータの末端バイトを格納している。

GET_part_send 関数、GET_part_rec 関数実行用のスレッドを立てる

```
406     # GETpartialコマンド
407     GET_set_s=[] #get sendをまとめて管理　スレッド用
408     GET_set_r=[] #get recをまとめて管理　スレッド用
409     for i in range(0,len(RouteTable)):
410         if(RouteTable[i][2] == 'none'):
411             # 直接GET要求
412             client_socket = socket(AF_INET, SOCK_STREAM)
413             client_socket.connect((server_name, server_port))
414         else:
415             client_socket = socket(AF_INET, SOCK_STREAM)
416             client_socket.connect((RouteTable[i][2], mid_port))
417         # 送受信用スレッド
418         GETs = threading.Thread(target=GET_part_send, args=(client_socket, \
419             RouteTable, token_str,server_file_name, sep_data_s[i], sep_data_e[i], i))
420         GET_set_s.append(GETs)
421         GETr = threading.Thread(target=GET_part_rec, args=(client_socket,sep_data_s))
422         GET_set_r.append(GETr)
423     start_time = time.time()
```

サーバと直接通信する経路の場合はサーバとのコネクションをそれ以外の経路の場合は 1 経由目のホストへのコネクションを行い、送信用のスレッド(GETs)と受信用のスレッド(GET_r)を立てる。それらのスレッドは GET_set_s, GET_set_r リストにすべて格納している。

GET_part_send 関数, GET_part_rec 関数を並列実行

```
425     for GETs in GET_set_s:# get sendを一斉に行う
426         GETs.start()
427
428     for GETr in GET_set_r:#get recを一斉に行う
429         GETr.start()
430
431     for GETr in GET_set_r:#get recが全部終わるまで待つ
432         GETr.join()
433     return start_time
434
```

GET_set_s, GET_set_r に格納したスレッドのタスクをそれぞれ一斉に行う。
このとき、GET_r スレッド(GET_part_rec 関数)が終わるまで処理を終了しないようにしている。返り値は GET_part_send を送り始めた時間。

・REP_cmd 関数

引数：RouteTable, サーバにあるファイル名

[1] サーバと直接通信する経路での SIZE コマンド

```
435  # REPコマンド
436  def REP_cmd(RouteTable, server_file_name):
437      # 直接REP要求
438      if(RouteTable[0][2] == 'none'):
439          # -------REPコマンド要求の送信---------
440          client_socket = socket(AF_INET, SOCK_STREAM)
441          client_socket.connect((server_name, server_port))
442          rep_msg = REP(server_file_name)
443          rep_msg += '\n'
444          client_socket.send(rep_msg.encode())
445
446          # -------REPコマンド応答の受け取り---------
447          REP_sentence = rec_res(client_socket)
448          end_time = time.time()
449          # print('REP_sentence', REP_sentence)
450          client_socket.close()
451
452          return end_time
```

REP 用の要求メッセージ作成しサーバ側に送信する。その後 REP コマンド応答メッセージをサーバから受け取る。返り値は REP コマンド応答文字列を受け取った時間。

[2] サーバと直接の通信する経路以外の経路での SIZE コマンド

REP 用パケットの作成

```
454      else:
455
456          client_socket = socket(AF_INET, SOCK_STREAM)
457          client_socket.connect((RouteTable[0][2], mid_port)) # 送信するホストとコネクション
458          """
459          REP用パケット
460          REP_pack = [クライアント名, 経由する1ホスト目, 経由する2ホスト目,
461                      サーバ, コマンドの種類(GET), コマンド要求文字列,
462                      経由回数, パケットの種類(Com), 要求(req) or 応答(rep),
463                      GET用のクライアントのポート番号]
464          """
465          REP_pack = f"{RouteTable[0][1]}/{RouteTable[0][2]}/{RouteTable[0][3]}\
466                      {RouteTable[0][4]}REP/{REP(server_file_name)}/{1}/Com/req/\
467                      {my_port_rep}"
468          REP_pack += '\n'
```

REP 用パケット

REP 用パケットの中身は

「クライアント名/経由する 1 ホスト目/経由する 2 ホスト目/サーバ名/コマンドの種類(REP)/コマンド要求文字列/経由回数/パケットの種類(Com)/送信用 or 応答用/REP 用のクライアントポート番号」となっている。

REP 用パケットの送信・REP コマンド応答文字列の受け取り

```
470         # print('REP_packet', REP_pack)
471         client_socket.send(REP_pack.encode()) # データ配列の送信
472 
473         # サーバからの応答の受け取り
474         REP_sentence = rec_res(client_socket)
475         end_time = time.time()
476         # print('REP_sentence', REP_sentence)
477         client_socket.close()
478 
479     return end_time
```

REP 用パケットの送信をし、REP 応答文字列を受け取る。返り値は REP コマンド応答文字列を受け取った時間

・main 関数

ping の実行, 1 経由目のホストの選択

```
481 if __name__ == '__main__':
482     # ---------ネットワークの状態を調べる--------------
483     print('-----routing-----')
484     # 全てのホストに対してPingを実行(並列処理)
485     packet_info = Ping_AllHost() # tupuleで受け取り(パケロス, rtt)
486     # 許容するパケットロスの上限(% 表示)
487     packet_loss_limit = 0
488     # clientからパケットを送るホストを決定する
489     select_ad_first(packet_info, packet_loss_limit)
```

Ping_ALLHost 関数、select_ad_first 関数を用いて全ホストへ Ping を行い。select_ad_first 関数で 1 経由目のホストを選択する。（サーバと直接通信する経路に対しては RouteTable 作成）

マルチスレッドを用いて並列に Route パケットを送信

```
491     # ThreadPoolExecutorでタイムアウトを実装している。
492     tpe = ThreadPoolExecutor(max_workers=5)
493     futures = []
494     TO_time = 60 # 保険で60秒でタイムアウトするように設定
495 
496     routing_1host() # 1ホスト経由のルーティング(関数をthread化)
497     routing_2host() # 2ホスト経由のルーティング(関数をthread化)
498     recv_Route_packet(TO_time) # threadの実行(パケットの受け取り)
```

routing_1host 関数、routing_2host 関数を用いて 1 ホスト経由、2 ホスト経由の選択可能な経路を用いた Route パケット送信タスクの実行キューを追加し、recv_Route_packet 関数で並列に実行。これらの返り値から RouteTable を作成している。

RouteTable のソート

```
500     # ----------ダウンロードしたファイルをルーティングした経路で送信----------
501     print('------download file------')
502
503     RouteTable = sorted(RouteTable) # timeによってソート
504
505     print()
506     print('sorted RouteTable:')
507     print(*RouteTable, sep='\n')
508     print()
```

RouteTable を 0 番目の列要素(指定した経路における ping による RTT の累積)で昇順にソートする。

SIZE コマンド

```
510     # SIZEコマンド
511     print()
512     print('SIZE Command')
513     # print('my_port', my_port)
514     data_size = SIZE_cmd(RouteTable)
515     print('data_size:',data_size)
516     print()
```

SIZE_cmd 関数で RouteTable の 0 行目の経路を用いて SIZE コマンドを行い、データサイズを受け取る。

最適経路選択(データサイズにより選択する経路本数を決定)

```
518     # GETで送るルート選択
519     # 何経路選択するか
520     RouteTables=RouteTable
521     if data_size < 1000*1024: #1M以下のものは経路を4つ
522         max_route = 4
523     elif data_size > 1000*1024*4:#4M以上のものは経路を1つ
524         max_route = 1
525     elif data_size > 1000*1024*3 and data_size <= 1000*1024*4:#3M以上のものは経路を2つ
526         max_route = 2
527     else: #その他のものは経路を3つ
528         max_route = 3
```

最適経路選択では RouteTable の上から何本の経路を選択するか決める。
まず、データサイズによって GET コマンドで選択する経路本数を以下のように仮決定する。
1MB 未満の場合は 4 経路選択
1MB 以上 3MB 未満の場合は 3 経路選択
3MB 以上 4MB 未満の場合は 2 経路選択
4MB 以上の場合は 1 経路選択

最適経路選択(RTT による除外)

```
    # あまりにも4つ採用した時に4つ目の経路のTTLが悪い場合は経路を3つにする
    if max_route >=4 and len(RouteTable) >= max_route:
        if 2*(RouteTable[2][0]-RouteTable[1][0])<(RouteTable[3][0]-RouteTable[2][0]):
            max_route = 3
    # あまりにも三つ採用した時に三つ目の経路のTTLが悪い場合は経路を二つにする
    if max_route>=3 and len(RouteTable) >= max_route:
        if 2*(RouteTable[1][0]-RouteTable[0][0])<(RouteTable[2][0]-RouteTable[1][0]):
            max_route = 2
    # あまりにも二つ経路を採用した時に二つ目のTTLが悪い場合は経路を一つにする
    if max_route>=2 and len(RouteTable) >= max_route and data_size > 1000*1024*1:
        if RouteTable[1][0]-RouteTable[0][0] > 400:
            max_route = 1

    if(len(RouteTable) >= max_route):
        tmp_RouteTable = RouteTable
        RouteTable = []
        for i in range(max_route):
            RouteTable.append(tmp_RouteTable[i])
```

RouteTable の各行の 0 番目の要素の値が他のものと比べてあまりにも大きい場合はその経路を選択しないようにする。
具体的には、まず RouteTable の n 行目(n≦3 n≠0)の経路がそれより上の n-1 本の経路と比べて RTT が大きいかどうかを見る。
これは、(n 本目の RTT – n-1 本目の RTT)が 2×(n-1 本目の RTT – n-2 本目の RTT)より大きいかどうかで判定をしている。大きい場合は最適経路として選択しない。また、n=1 のときは(n 本目の RTT – n-1 本目の RTT)が 400[ms]より大きい場合は選択しない。

データ分割・GET_PARTIAL コマンド

```
    print('GET Command')
    # print('my_port', my_port)
    # GETpartialコマンド
    start_time = GET_part_cmd(RouteTable, token_str, server_file_name, data_size)
    # print()

```

GET_part_cmd関数でデータファイルサイズを選択した最適経路における ping による RTT の累積の逆数の比で分割し、さらに GET_PARTIAL コマンドの実行をする。返り値として GET_PARTIAL 要求文字列またはパケットを送信した時間を返している

REP コマンド

```
    # print('REP Command')
    end_time = REP_cmd(RouteTable,server_file_name)
    print(f'time {end_time - start_time}')
```

REP_cmd 関数で RouteTable の 0 行目の経路を用いて REP コマンドを行い、返り値として REP 応答文字列を受け取った時間を受け取る。最後に GET_partcmd 関数と REP_cmd 関数の返り値からクライアント側で競技にかかったおおよその時間を計測している。

○ mid_p2s.py

・rec_res 関数

引数：ソケット

```
48  # \nまでの文字列の受け取り(\nを含まない)
49  def rec_res(soc):
50      # 応答コードの受け取り
51      recv_bytearray = bytearray() # 応答コードのバイト列を受け取る配列
52      while True:
53          b = soc.recv(1)[0]
54          if(bytes([b]) == b'\n'):
55              rec_str = recv_bytearray.decode()
56              print(rec_str)
57              break
58          recv_bytearray.append(b)
59      # print('received')
60
61      return rec_str
```

コネクションしているホストから’¥n’まで文字列を受け取る。

・Ping_mid 関数

転送管理サーバが行う ping のパラメータ

引数：ping を行う宛先のホスト名，許容パケットロス率

```
40  # ping
41  def Ping_mid(ad, p_loss_lim):
42      flg = False
43      # 正規表現'%'が後ろにつく[0,100]の数字を検索するための正規表現オブジェクトを生成
44      regex = re.compile(r'\d{1,3}(?=%)')
45      # ping -c 10 -w 1000 adrress
46
47      ping = subprocess.run(
48          ["ping", "-c", "10","-i", "0.2","-s","65507","-q", ad],
49          stdout=subprocess.PIPE,     # 標準出力は判断のため保存
50          stderr=subprocess.DEVNULL # 標準エラーは捨てる
51      )
```

指定したホストに対して ping を行う。ping は 65507B を 0.2 秒間隔で 20 回送信するようパラメータを設定する。

ping 出力の受け取り・パケットロス率と RTT の抜出

```
52
53      output = ping.stdout.decode("cp932")
54      # print(output)
55
56      # outputからpacketlossを取り出すする
57      packet_loss = regex.search(output)
58      # pingコマンドが成功->パケットロス率を返す,失敗->パケットロス率を100%とする
59      if packet_loss == None:
60          return_p_loss = 100
61      else:
62          return_p_loss = float(packet_loss.group())
63
64      if(return_p_loss <= p_loss_lim):
65          # outputからrttの平均を取り出す
66          i = output.find('rtt')
67          rtt_info = output[i:]
68          rtt_info = re.split('[/ ]', rtt_info)
69          # print(rtt_info)
70          rtt = float(rtt_info[7])
71          flg = True
72      else:
73          rtt = 100000000
74
75      print()
76      print('ping to',ad)
77      print('packetloss:',return_p_loss,'rtt',rtt, flg)
78      print()
79
80      return flg, rtt
```

実行した ping の出力文字列からパケットロス率と RTT の抜出を行う。
パケットロス率が許容範囲内であれば ping フラグを True に範囲外であれば ping フラグを False にする。返り値として ping フラグ, RTT を返す。

・send_packet 関数

引数：ソケット, Route 用パケット配列

```
82  # Routeパケットの送信に使う
83  def send_packet(soc, pack):
84      # pack_array = pack #
85      pack_str = '/'.join(map(str,pack)) # 配列の要素を/で区切り、文字列にする
86      pack_str += '\n'
87
88      print()
89      print('send_packet')
90      print(pack_str)
91      print()
92
93      soc.send(pack_str.encode()) # データ配列の送信
94      # soc.close()
```

引数として渡されたパケット配列を'/'区切りの文字列に変換し指定したホストへ送信する。

・fix_route_packet 関数

引数：Route パケットの配列

```
96  # routeパケットを適切な型に変換
97  def fix_route_packet(pack):
98      if(pack[4] == 'False'):
99          pack[4] = False
100     elif(pack[4] == 'True'):
101         pack[4] = True
102
103     pack[5] = int(pack[5])
104     pack[6] = int(pack[6])
105     pack[9] = int(pack[9])
106     pack[10] = float(pack[10])
```

Route パケットの配列の各要素を文字列から適切な型に変換する。

・relay_packet 関数

引数：ソケット

```
130 def relay_packet(connect_soc):
131     pack_string = rec_res(connect_soc) #  パケット(文字列区切り)受け取り
132     print()
133     print('recv_packet')
134     print(pack_string)
135     print()
136     # /で区切ってリストに格納(各要素は文字列として認識される→intやboolに変換が必要)
137     pack = pack_string.split('/')
```

相手ホストから送られてきたパケット文字列を受け取り、パケット配列に変換する。

[1] 送られてきたパケットが Route 用パケットだった場合

```
142     # -----Routing用のパケットだった場合-----
143     if(pack[7] == 'Route'):
144         """
145         Route用パケット
146         pack = [クライアント名, 経由する1ホスト目, 経由する2ホスト目, サーバ,
147                 パケット到達性フラグ, TTL, 経由回数, パケットの種類(Route),
148                 送信用(req) or 応答用(rep), クライアントのポート番号
149                 , 指定した経路でのrttの累積]
150         """
151
152         # パケットの要素を適切な型に変換
153         pack = fix_route_packet(pack)
154         send_pack = pack # 送信用パケット
```

153 行目で Route パケットの配列の各要素を文字列から適切な型に変換、
154 行目でパケットの中身を送信用パケットにコピーをしている。

[1-1] 送られてきたパケットをみて TTL=2 のとき

```
155
156         if(pack[8] == 'req'):
157             # TTL(pack[5])==2ならば、転送管理サーバへ送信
158             if(pack[5] == 2):
159                 mid_name = pack[pack[6]+1]
160                 flg_ping, rtt = Ping_mid(mid_name, 0)
```

パケットの経由回数+1 番目の要素(この場合は 2 番目の要素、つまり 2 経由目のホスト名)を取り出し、そのホストに対して ping を実行する。

[1-1-1] ping のパケットロス率が許容範囲内だった場合

```
162                 if(flg_ping):
163                     send_pack[6] += 1 # relay_numをインクリメント
164                     send_pack[5] -= 1 # TTLをデクリメント
165                     # 経由するホストが増えるのでrelay_num(info_pack[6])をインクリメント
166                     send_pack[10] += rtt
167                     soc_to_mid = socket(AF_INET, SOCK_STREAM)
168                     soc_to_mid.connect((mid_name, mid_port))
169                     send_packet(soc_to_mid, send_pack)
170 
171                     # 転送管理サーバからの受け取り
172                     pack_string = rec_res(soc_to_mid) #  パケット(文字列)受け取り
173 
174                     # /で区切ってリストに格納(各要素は文字列として認識される→intやboolに変換が必要)
175                     pack = pack_string.split('/')
176                     pack = fix_route_packet(pack)
177                     # connection_socketへ送信
178                     send_pack = pack
179                     send_pack[6] -= 1
180                     send_packet(connect_soc, send_pack)
181                     soc_to_mid.close()
```

163 行目で送信用パケットの経由回数をインクリメント
164 行目で送信用パケットの TTL をデクリメント
165 行目で送信用パケットの 10 番目の要素に ping から得られた RTT を加算し
パケットの指定したホスト(パケットの 2 経由目のホスト)へパケットを送信する。
さらに、そのコネクションホストから返答パケットを受け取る。
返答パケットを受け取ったら
175 行目で受け取ったパケット文字列を配列に変換し
176 行目で適切な型に変換
178 行目で送信用パケットに送られてきたパケットをコピー
179 行目で送信用パケットの経由回数をデクリメント
最後に保持していたクライアントからのコネクションを用いてクライアントへ送信用パケットを送信する。

[1-1-2] ping のパケットロス率が許容範囲外だった場合

```
183                 else:
184                     send_pack[6] -= 1
185                     send_pack[4] = False
186                     send_pack[8] = 'rep'
187                     send_packet(connect_soc, send_pack)
```

184 行目で送信用パケットの経由回数をデクリメント
185 行目で送信用パケットの到達可能フラグを False
そして、保持していたクライアントからのコネクションを用いてクライアントへ送信用パケットを送信する。

[1-2] 送られてきたパケットをみて TLL=1 のとき

```
189             # TTL(info_pack[5])==1ならばTTLを1つ減らして、サーバと同じ名前の転送管理サーバへping
190             # その後pingがタイムアウトorパケットロスしなければroute/パケットを転送管理サーバへ送信
191             elif(pack[5] == 1):
192                 flg_ping, rtt = Ping_mid(server_name, 0)
```

サーバに対して ping を行う。

[1-2-1] ping のパケットロス率が許容範囲内だった場合

```
194             # pingで良い経路であると分かったらrttを加えて1コ前に戻る
195             if(flg_ping):
196                 send_pack[6] -= 1
197                 send_pack[8] = 'rep'
198                 send_pack[5] -= 1 # TTLをデクリメント
199                 send_pack[10] += rtt
200                 if(send_pack[send_pack[6]] == cl_name):
201                     cl_port = send_pack[9]
202                     # soc_to_cl = socket(AF_INET, SOCK_STREAM)
203                     # soc_to_cl.connect((cl_name, cl_port))
204                     send_packet(connect_soc, send_pack)
205                 else:
206                     mid_name = send_pack[send_pack[6]]
207                     # soc_to_mid = socket(AF_INET, SOCK_STREAM)
208                     # soc_to_mid.connect((mid_name, my_port))
209                     send_packet(connect_soc, send_pack)
```

196 行目で送信用パケットの経由回数をデクリメント
198 行目で送信用パケットの TTL をデクリメント
199 行目で送信用パケットの 10 番目の要素に ping から得られた RTT を加算している。
指定した経路上で自分より一個前のホストがクライアントだった場合クライアントへ
転送管理サーバだった場合は転送管理サーバへパケットを送信用パケットを送信する
(コネクションしている相手ホストへ送信するだけなので if 文で分ける必要はなかった)

[1-2-2] ping のパケットロス率が許容範囲外だった場合

```
211                 # 悪い経路だったら到達性をFalseにして1コ前に戻る
212                 else:
213                     send_pack[6] -= 1
214                     send_pack[4] = False # パケットが正当な経路で送られなかったのでFalse
215                     send_pack[8] = 'rep'
216                     send_pack[5] -= 1 # TTLをデクリメント
217 
218                     if(send_pack[send_pack[6]] == cl_name):
219                         cl_port = send_pack[9]
220                         # soc_to_cl = socket(AF_INET, SOCK_STREAM)
221                         # soc_to_cl.connect((cl_name, cl_port))
222                         send_packet(connect_soc, send_pack)
223 
224                     else:
225                         mid_name = send_pack[send_pack[6]]
226                         # soc_to_mid = socket(AF_INET, SOCK_STREAM)
227                         # soc_to_mid.connect((mid_name, my_port))
228                         send_packet(connect_soc, send_pack)
```

213 行目で送信用パケットの経由回数をデクリメント
214 行目で送信用パケットの到達可能フラグを False
216 行目で送信用パケットの TTL をデクリメント
指定した経路上で自分より一個前のホストがクライアントだった場合クライアントへ
転送管理サーバだった場合は転送管理サーバへパケットを送信用パケットを送信する
(コネクションしている相手ホストへ送信するだけなので if 文で分ける必要はなかった)

[2] 送られてきたパケットが Com(コマンド)用パケットだった場合

```
230     # ----コマンド用のパケットだった場合
231     elif(pack[7] == 'Com'):
232         """
233         Com用パケット
234         pack = [クライアント名, 経由する1ホスト目, 経由する2ホスト目,
235                 サーバ, コマンドの種類, コマンド要求文字列,
236                 経由回数, パケットの種類(Com), 要求(req) or 応答(rep),
237                 コマンド用のクライアントのポート番号]
238         """
239         pack = fix_com_packet(pack)
240         send_pack = pack
```

相手ホストから送られてきたパケット文字列を受け取り、パケット配列に変換する。
さらに送られてきたパケットを送信用パケットへコピーする。

[2-1] 指定した経路が 1 ホスト経由だった場合

```
242         if(pack[8] == 'req'): # パケットが要求用
243             # 経路が1ホスト経由だった場合
244             if(pack[pack[6]+1] == 'none'): # pack[6](経由数) == 1である
245                 send_pack[6] += 1 # 経由数をインクリメント
246                 # ----サーバとのやり取り(コマンド要求・受け取り)--------
247                 soc_to_ser = socket(AF_INET, SOCK_STREAM)
248                 soc_to_ser.connect((server_name, server_port))
249                 req_sentence = pack[5]
250                 req_sentence += '\n'
251                 soc_to_ser.send(req_sentence.encode())
252                 sentence = rec_res(soc_to_ser) # コマンド応答
253                 print(sentence)
254                 sentence += '\n'
255 
256                 connect_soc.send(sentence.encode()) # クライアント側へ応答を返す
257                 if(pack[4] == 'GET'):
258                     # print('received server file')
259                     while True:
260                         b = soc_to_ser.recv(1024)
261                         connect_soc.send(b)
262                         if(len(b) <= 0):
263                             break
264                 soc_to_ser.close()
```

送られてきたパケットからコマンド要求文字列を取り出しサーバへコマンド要求を送る。サーバから応答文字列が返ってきたら保持していたクライアントへのコネクションを用いて応答文字列をクライアント側へ送信する。コマンドの種類が GET だった場合はサーバからファイルの中身を受け取りながら保持していたクライアントへのコネクションを用いてファイルの中身をクライアント側へ送る。

[2-2] 指定した経路が 2 ホスト経由だった場合

[2-2-1] (経由回数)=1 だった場合

```
266             else:
267                 # 転送管理サーバへパケットを送信
268                 if(pack[6] == 1):
269                     send_pack[6] += 1 # 参照番号をインクリメント
270                     mid_name = send_pack[send_pack[6]]
271                     soc_to_mid = socket(AF_INET, SOCK_STREAM)
272                     soc_to_mid.connect((mid_name, mid_port))
273                     send_packet(soc_to_mid, send_pack)
274 
```

269 行目で送信用パケットの経由回数をインクリメント

送信用のパケットの経由回数番目の要素をホスト名とするホストへコネクション(指定した経路の自ホストの次のホスト)

指定したホストへ送信用パケットのパケットを送信する。

```
274
275                 # 転送管理サーバからの受け取り
276                 pack_string = rec_res(soc_to_mid) #  パケット(文字列)受け取り
277                 # /で区切ってリストに格納(各要素は文字列として認識される→intやboolに変換が必要)
278                 pack = pack_string.split('/')
279                 pack = fix_com_packet(pack)
280                 send_pack = pack
281                 # クライアント側へファイルを送信
282                 sentence = pack[5]
283                 sentence += '\n'
284                 connect_soc.send(sentence.encode())
285                 if(pack[4] == 'GET'):
286                     # print('received server file')
287                     while True:
288                         b = soc_to_mid.recv(1024)
289                         connect_soc.send(b)
290                         if(len(b) <= 0):
291                             break
```

さらに、そのコネクションホストから返答パケットを受け取る。
返答パケットを受け取ったら
278 行目で受け取ったパケット文字列を配列に変換し
279 行目で適切な型に変換
送られてきたパケットからコマンド応答文字列を取り出し、保持していたクライアントへのコネクションを用いて応答文字列をクライアント側へ送信する。コマンドの種類が GET だった場合は転送管理サーバからファイルの中身を受け取りながら保持していたクライアントへのコネクションを用いてファイルの中身をクライアント側へ送る。

[2-2-2] (経由回数)=2 だった場合

```
293             elif(pack[6] == 2):
294                 send_pack[6] -= 1
295                 # ----サーバに対するコマンド要求・受け取り--------
296                 soc_to_ser = socket(AF_INET, SOCK_STREAM)
297                 soc_to_ser.connect((server_name, server_port))
298                 req_sentence = pack[5]
299                 req_sentence += '\n'
300                 soc_to_ser.send(req_sentence.encode())
301                 sentence = rec_res(soc_to_ser)
302
303                 send_pack[5] = sentence
304                 send_packet(connect_soc, send_pack)
305
306                 if(pack[4] == 'GET'):
307                     # print('received server file')
308                     while True:
309                         b = soc_to_ser.recv(1024)
310                         connect_soc.send(b)
311                         if(len(b) <= 0):
312                             break
313                 soc_to_ser.close()
```

送られてきたパケットからコマンド要求文字列を取り出し、サーバへコマンド要求を送る。サーバからコマンド応答文字列が返ってきたら、送信用パケットの 5 番目の要素をコマンド応答文字列に書き換え、保持していた転送管理サーバへのコネクションを用いて送信用

パケットを転送管理サーバ側へ送信する。さらに、コマンドの種類が GET だった場合はサーバからファイルの中身を受け取りながら保持していた転送管理サーバへのコネクションを用いてファイルの中身を転送管理サーバ側へ送る。

・main 関数

```
321  def main():
322      # -----転送管理サーバを経由してサーバとクライアントの通信をする----
323      my_socket = socket(AF_INET, SOCK_STREAM)
324      my_socket.bind(('', my_port)) # 自身のポートをソケットに対応づける
325      my_socket.listen(5)
326
327      print('The server is ready to receive info packet')
328
329      while True:
330          # クライアントからの接続があったら、それを受け付け、
331          # そのクライアントとの通信のためのソケットを作る
332          connection_socket, addr = my_socket.accept() # 送信元ホストとのコネクション
333          client_handler = threading.Thread(target=relay_packet, args=(connection_socket,))
334          client_handler.start()  # スレッドを開始
```

コネクション待ち受けのスレッドを相手側ホストからコネクションされるたびに立てて並列に relay_packet 関数を実行している。

2.3 プログラムの具体的動作

○ client_p2s.py(クライアント側)

クライアント側のプログラムの具体的な動作は以下の通り。

① クライアントから全ホストに対して ping を実行し、クライアントと 1 ホスト目との間の経路のパケットロス率を調べる。(Ping 関数, Ping_ALLHost 関数)

② クライアントとの間のパケットロス率が 0%であるホスト名をリストに格納する。(ad_first リスト)ここで、サーバと直接通信する経路のパケットロス率 0%だった場合 RouteTable にその経路情報とその経路に対しての ping から得られた RTT を格納する。(select_ad_first 関数)

③ 各経路に対しての Route 用パケットを作成・送信をする。ここで、選択する経路の 1 ホスト目は②で作成した ad_first リストから選択する。
1 ホスト経由の場合は TTL=1 に、2 ホスト経由の場合は TTL は 2 に設定し、1 ホスト、2 ホスト経由の Route 用パケット送信をマルチスレッドを用いて並列に実行している。(routing_1host 関数, ruouting_2host 関数, exchange_Routepacket 関数)

④ 返答用の Route 用パケットを受け取り、その経路が到達可能であった場合は RouteTable に経路情報と Ping による指定した経路での累積 RTT を格納する。(recv_Route_packet 関数)

⑤ 作成した RouteTable を Ping による指定した経路での累積 RTT で昇順にソートする。

⑥ RouteTable の一番上の経路(累積 RTT の最も小さい経路)を用いて SIZE コマンドを行いデータサイズを受け取る(SIZE_cmd 関数)

⑦ データサイズと RouteTable から最適経路の選択を行う。

⑧ 選択した最適経路それぞれで ping による累積 RTT の逆数の比で受け取るファイルサイズを分割する。さらにその分割ファイルサイズで GET_PARTIAL コマンドを行うための GET 用パケットまた GET 要求コマンド送信し、GET 応答コマンドとファイルの中身を受け取る。(GET_part_send 関数, GET_part_rec 関数, GET_part_cmd 関数)

⑨ RouteTable の一番上の経路(累積 RTT の最も小さい経路)を用いて REP コマンドを行い REP 応答文字列受け取り時間を返り値として受け取ってプログラム終了(REP_cmd 関数)

○ mid_p2s.py
転送管理サーバの具体的な動作は以下の通り。
[1] Route 用パケットが送られてきた場合
TTL が 2 だった場合と TTL が 1 だった場合の 2 つの異なる処理を行う。

[1-1] TTL が 2 だった場合
① 指定した経路における経由する 2 ホスト目に対して ping を行う。

② ping のパケットロス率が許容範囲内の場合は Route パケットの中身の経由回数をインクリメント、TTL をデクリメント、Route パケットの 10 番目の要素に ping から得られた RTT を加算して Route パケットを転送管理サーバ側へ送信する。ping のパケットロス率が許容範囲外だった場合は到達可能フラグを False にしてクライアント側へ Route パケットを返す

③ ping のパケットロス率が許容範囲内だった場合は、さらに転送管理サーバ側から返答パケットを受け取る。返答パケットの中身の経由回数をデクリメントしたらクライアントへ送信用パケットを送信する。

[1-2] TTL が 1 だった場合
① サーバへ ping を行う

② ping のパケットロス率が許容範囲内の場合は Route パケットの中身の経由回数をデクリメント、TTL をデクリメント、Route パケットの 10 番目の要素に ping から得られた RTT を加算してして Route パケットを指定した経路の自分から一つ前のホストへ送信する。
ping のパケットロス率が許容範囲外だった場合は到達可能フラグを False にして Route パケットを指定した経路の自分から一つ前のホストへ送信する。

[2] Com(コマンド)用パケットが送られてきた場合
経路が 1 ホスト経由だった場合と 2 ホスト経由だった場合の 2 つの異なる処理を行う。

[2-1] 経路が 1 ホスト経由だった場合
① 送られてきた Com(コマンド)パケットからコマンド要求文字列を取り出しサーバへコマンド要求を送る。

② サーバから応答文字列が返ってきたらそれををクライアント側へ送信する。コマンドの種類が GET だった場合はサーバからファイルの中身を受け取りながらファイルの中身をクライアント側へ送る。

[2-2] 経路が 2 ホスト経由だった場合

[2-2-1] (経由回数) = 1 だった場合
① 受け取った Com パケットの中身の経由回数をインクリメントして、指定した経路における自ホストの次のホストへパケットを送信する。

② パケットを送信したホストから返答パケットを受け取る。

③ 返答パケットを受け取ったら、パケットからコマンド応答文字列を取り出し、これをクライアントへ送信する。コマンドの種類が GET だった場合は返答パケットを受け取ったホストからファイルの中身を受け取りながらこのファイルの中身をクライアント側へ送る。

[2-2-2] (経由回数) = 2 だった場合

① 送られてきたパケットからコマンド要求文字列を取り出し、サーバへコマンド要求を送る。

② サーバからコマンド応答文字列が返ってきたら 送信用パケットの 5 番目の要素をコマンド応答文字列に書き換え、Com パケットが送られてきたホストへパケットを送信する。コマンドの種類が GET だった場合はサーバからファイルの中身を受け取りながらこのファイルの中身をパケットが送られてきたホストへ送る。

2.4 システムの動かし方

転送管理側のプログラムを先にすべて動かし、その後クライアント側のプログラムを動かす。それぞれの実行手順は以下の通り。

○ 転送管理側

コマンドライン上で

python3 mid_p2s.py

と入力して実行

○ クライアント側

コマンドライン上で

python3 client_p2s.py (サーバ名) (サーバにあるファイル名) (トークン文字列)

と入力して実行

2.5 プログラム実装上の工夫

工夫した点は以下の通り。

○ プログラムが分かりやすいようコメントを適宜挿入した。

○ プログラムが分かりやすいようできるだけプログラムの処理を関数化させた。

○ プログラムが分かりやすいようにできるだけ変数名を適切な名前にした。

○ 各処理をできるだけ並列化をして競技時間を短縮できるようにした

○ TCP のコネクションを最小限にした。

3. 動作の結果

本番の成績

課題 1 :クライアント pbl5a　サーバ pbl1a

ダウンロードするファイル 3MB dat ファイル　　ダウンロード時間 6.94[sec]

課題 2:クライアント pbl4a　サーバ pbl6a

ダウンロードするファイル 640KB dat ファイル　　ダウンロード時間 3.68[sec]

捕捉データ（昨年の AWS 上の環境でクライアントで GET を送り、REP の返信がサーバから来るまでの時間）
①クライアント pbl1a　サーバ pbl3a
ダウンロードするファイル rnd50k.dat　　ダウンロード時間 3.0[sec]
②クライアント pbl1a　サーバ pbl2a
ダウンロードするファイル rnd1M.dat　　ダウンロード時間 6.9[sec]
③クライアント pbl1a　サーバ pbl2a
ダウンロードするファイル rnd5M.dat　　ダウンロード時間 18.4[sec]

4. 考察

私たちの班では経路探索に ping とパケットの送信を使用している。この経路探索方法で出来るのは輻輳に弱い経路を除外すること、パケットロスが起こる経路を除外することで、「経路の帯域幅がどのくらいであるのか」については正確に知ることはできない。
よって Ping の RTT を利用して大体の経路の帯域幅の優劣をつけ、優秀なものから順にファイルの受信で使う経路とする厳選を行った。
RTT の逆比をとってダウンロードするファイルのデータ分割を行いたく、理想的にはデータを分割すればするほどダウンロード時間は短縮できるはずだった。
しかし、データのダウンロードに使う経路が増えれば増えるほどダウンロード時間の短縮ができたかというとそうではない。

以下に 3 つの例を示す。
①クライアント pbl1a　サーバ pbl3a　ダウンロードするファイル rnd50k.dat

一つ目　転送管理サーバ	二つ目　転送管理サーバ	RTT [ms]
pbl4a	NULL	1158
pbl6a	pbl4a	1776
pbl6a	NULL	1843

これが経路探索で得られた経路表であるが、この三つの経路を全て使ってダウンロードする場合と、上から二つ、一つを使ってダウンロードする場合でダウンロード時間の違いについて調べた。

経路の数	ダウンロード時間[s]
3	3.0
2	3.4
1	3.5

この例では経路を増やせば増やすほどダウンロード時間が短縮できていることがわかる。

②クライアント pbl1a　サーバ pbl2a　ダウンロードするファイル rnd1M.dat

一つ目　転送管理サーバ	二つ目　転送管理サーバ	RTT [ms]
pbl4a	pbl3a	1364
pbl6a	pbl5a	1516
pbl6a	pbl7a	1887

これも同様に経路の数とダウンロード時間について調べる。

経路の数	ダウンロード時間[s]
3	7.6
2	6.9
1	7.0

なんとこれでは経路を二つだけしか採用しない時が最もダウンロード時間の短縮が出来た。この時点ですでに経路を増やせば増やすほどダウンロード時間の短縮が出来るという仮説は破られた。

③クライアント pbl1a　サーバ pbl2a　ダウンロードするファイル rnd5M.dat

一つ目　転送管理サーバ	二つ目　転送管理サーバ	RTT [ms]
pbl4a	pbl3a	1365
pbl6a	pbl5a	1517
pbl6a	pbl7a	1888

これも同様に経路の数とダウンロード時間について調べる。

経路の数	ダウンロード時間[s]
3	26.0
2	19.4
1	18.4

これは②と同じ経路で調べてファイルのサイズを変更した場合であるが、③の条件では経路を一つのみ採用した場合が最もダウンロードが早いことがわかった。

なぜ②、③のようなことが起こったのか、原因は使用する経路の帯域幅を調べることが出来なかったため、輻輳制御の起こるデータの量を経路に流してしまったことが原因であると

考えた。
公開されている AWS 上の環境を基に経路の帯域幅を考える。
①の例では帯域幅は以下のようになっている。

一つ目　転送管理サーバ	二つ目　転送管理サーバ	経路全体の帯域幅[bps]
pbl4a	NULL	3M
pbl6a	pbl4a	2.5M
pbl6a	NULL	2M

大体経路の帯域幅が太い順に経路表の作成が出来ていることがわかる。①の例では分割した時のファイルの大きさに比べて経路の帯域幅が十分に大きかったため、使用する経路が多ければ多いほどダウンロード時間が短くなったのだろう。

②、③の例では帯域幅は以下のようになっている。

一つ目　転送管理サーバ	二つ目　転送管理サーバ	経路全体の帯域幅[bps]
pbl4a	pbl3a	3M
pbl6a	pbl5a	2.5M
pbl6a	pbl7a	2M

ここで pbl1a と pbl6a の帯域幅が 2.5M であることに注意したい。②では 1Mbyte のファイルを、③では 5Mbyte のファイルをダウンロードするのだが、おそらく②、③で経路を三つ採用してダウンロードすると遅くなるのは pbl1a と pbl6a の経路でダウンロードするファイルのサイズが大きくなってしまい、輻輳が起こっているからだと考えられる。
これで②が経路二つを採用した時に最速でダウンロードできる理由がわかったが、③で経路一つのみを採用した時に最速でダウンロードできる理由はわかっていない。
③で経路一つのみ採用した時に最速でダウンロードできるのは何が原因であるのかとして考えたのが輻輳制御による転送量の変化である。
きっと③のケースでは経路を一つだけ使用した場合も、経路を二つ使用した場合も輻輳は起こっていると考えた。経路を一つだけ使用した場合の方が経路を二つ使用した場合よりも経路あたりの輻輳の数が少なかったのではないかと思う。だから大きい帯域幅の経路一つのみを使ってダウンロードするのが最速だったのではないだろうか。

このように正確な帯域幅を調べて知れなかったがゆえに経路の厳選をしなければいけなくなった。経路の厳選は Ping で得た RTT を基準に行ったのだが、もっと早く競技の時間をするためには帯域幅を正確に知るシステムと各経路の帯域幅に考慮したデータ分割が出来ればよかったと思った。

5. 参考文献

・基本から学ぶ　TCP と輻輳制御 …押さえておきたい輻輳制御アルゴリズム
　第 1 回輻輳制御とは何か
著者：中山悠
URL： https://gihyo.jp/admin/serial/01/tcp-cc/0001 (2022/1/18 アクセス)

・[Python]Ping で死活監視を行う
著者：しげっち
URL：https://shigeblog221.com/python-ping/(2022/1/9 アクセス)

・Python で concurrent.futures を使った並列タスク実行
著者：tag03
URL：https://qiita.com/tag1216/items/db5adcf1ddcb67cfefc8

・通信速度に影響する？帯域幅とは
URL：https://mvno.freebit.com/column/iot-m2m/bandwidth.html

・コンピュータネットワーク概論
出版社：共立出版
著者：水野忠則，奥田隆史，中村嘉隆，井手口哲夫，田学軍，清原良三，石原進，
　　久保田信一郎，勅使河原海，岡崎直宜
出版年：2014 年(初版第 1 刷発行)