109550073 陳宥安

Network System Capstone
Homework 5
Report

1. **Explain how you implement error control: 5%**

簡而言之就是每個packet都有ACK並且有timer的功能:

```
#! ERROR control
#? Resend timeout
self.time_resend = 2
self.init_time_resend = self.time_resend
```

```
#? Send the window data
for i in range(window_head, window_end):
    send_packet = struct.pack("@iiii", 1, stream_id, packet_num, i)
    send_packet += datas[i]
    self.sock.sendto(send_packet, self.client_addr)
    self.recv_info[stream_id][1][i] = time.time() + self.time_resend
```

發送端每送一個packet時，會設定他的timer，並且會等待他的ACK，要packet的timer時間前收到ACK，才算是成功傳送了該packet，否則就會從沒收到ack最小的index開始resend packet，當發送端確認stream中的每個packet都有收到ack後就算傳送完畢。

接收端的話，將每個收到的packet放到buffer(tmp_rcv_buffer)暫存並回傳ACK:

```
#? send back ACK
send_packet = struct.pack("@iiii", 2, recv_stream_id, recv_packet_size, recv_packet_index)
send_packet += recv_buf
self.sock.sendto(send_packet, self.client_addr)
```

當接收端收到stream的所有packet後，依照packet上標記的packet_index重組stream data並將其放入self.recv_buffer:

```
#? if received all stream data -> concate them and send recv_buffer, waitting to be retrieved by self.recv()
if len(tmp_recv_buffer[recv_stream_id]) == recv_packet_size:
    # print("stream_id", recv_stream_id, "get all data", len(tmp_recv_buffer[recv_stream_id]), recv_packet_size)
    tmp_data = bytearray()
    for i in range(len(tmp_recv_buffer[recv_stream_id])):
        tmp_data += tmp_recv_buffer[recv_stream_id][i]
    self.mutex_buffer.acquire()
    self.recv_buffer.append((recv_stream_id, tmp_data))
    self.mutex_buffer.release()
    del tmp_recv_buffer[recv_stream_id]
```

這樣就達到了error control的效用(確保packet一定有傳到接收端)

2. **Explain how you implement flow control: 5%**

先設定自身self.recv_buffer最大的大小(這裡預設5000 bytes)

```
#! FLOW control
#? recv_buffer max size
self.max_recvbuffersize = 5000
#? overflow flags for my and receiver's recv_buffer
self.my_recv_overflow = False
self.recv_overflow = False
```

每當tmp_recv_buffer收齊整個stream的packets時並寫入self.recv_buffer時，程式會計算self.recv_buffer中各個stream data的總大小，如果大於self.max_recvbuffersize的話，接收端會標注flag(self.my_recv_overflow)並告訴sender自己已經overflow:

```
#? Calculate self.recv_buffer size
cur_recvbuffersize = 0
if len(self.recv_buffer) > 0:
    for recv_tuple in self.recv_buffer:
        #? 16 -> header size of each packet
        cur_recvbuffersize += (len(recv_tuple[2]) + 16*recv_tuple[1])
# print("cur recvbuffer size", cur_recvbuffersize)
#? check if my recv_buffer overflow
if cur_recvbuffersize >= self.max_recvbuffersize and self.my_recv_overflow == False:
    # print("my recv buffer overflow!")
    self.send(-1, b'overflow')
    self.my_recv_overflow = True
```

如果後續有用self.recv()把stream data移出self.recv_buffer時，程式會再檢查這時的self.rcv_buffer的大小，如果小於self.max_recvbuffersize的話，接收端會標注flag(self.my_recv_overflow)並告訴sender自己又available了:

```
#? Calculate self.recv_buffer size
cur_recvbuffersize = 0
if len(self.recv_buffer) > 0:
    for recv_tuple in self.recv_buffer:
        #? 16 -> header size of each packet
        cur_recvbuffersize += (len(recv_tuple[2]) + 16*recv_tuple[1])
# print("cur recvbuffer size", cur_recvbuffersize)
#? check if my recv_buffer become available again
if cur_recvbuffersize < self.max_recvbuffersize and self.my_recv_overflow == True:
    self.send(-2, b'available')
    self.my_recv_overflow = False
```

發送端的部份，如果收到接收端 buffer overflow的訊息的話，就設定flag(self.recv_overflow)並停止傳新的data給接收端，直到接收端送buffer available的訊息後才可以恢復傳送，要不然send會卡在原地:

```
#? stream_id = -1 -> receiver recv_buffer overflow
if recv_stream_id == -1 and self.recv_overflow == False:
    # print("!!! recvbuffer overflow  !!!")
    self.recv_overflow = True
#? stream_id = -2 -> receiver recv_buffer available
elif recv_stream_id == -2 and self.recv_overflow == True:
    # print("!!! recvbuffer available  !!!")
    self.recv_overflow = False
```

這樣就達到了flow control的效用(接收端overflow了就別再傳了，除非又有空間)

3. **Explain how you implement congestion control: 5%**

簡而言之就是在UDP刻出Sliding Window的功能，首先先設定雙方socket recvfrom的大小，client會在建立連線時傳給server以統一大小，再來self.expect_ackrate是指sliding window縮小的臨界值，最後再設定self.windowsize:

```
#! CONGESTION control
#? socket.recvfrom size for both server and client
self.recvsize = 30
#? expect received_ack rate[0:1]
self.expect_ackrate = 0.5
#? Sliding window size
self.windowsize = 3
```

發送端將以self.recvsize為標準將超過此大小的stream data切成多個packet，第一次傳送時以self.windowsize個封包傳送:

```
#? Slice data
datas = []
max_datasize = self.recvsize - 16
if len(data) > max_datasize:
    i = 0
    while len(data) >= max_datasize*(i):
        datas.append(data[max_datasize*i : max_datasize*(i+1)])
        i+=1
else:
    datas.append(data)
packet_num = len(datas)
# print("stream_id", stream_id, "total packet num", packet_num)
# print("packets", datas)
#? INIT and Send first window data
window_head = 0
window_end = min(packet_num, self.windowsize)
self.recv_info[stream_id][0] = [False for x in range(packet_num)]
self.recv_info[stream_id][1] = [0 for x in range(packet_num)]
for i in range(window_end):
    send_packet = struct.pack("@iiii", 1, stream_id, packet_num, i)
    send_packet += datas[i]
    self.sock.sendto(send_packet, self.client_addr)
    self.recv_info[stream_id][1][i] = time.time() + self.time_resend
```

而發送端會根據接收端回覆的ack數量來調整self.windowsize，如果window內所有packet都收到ACK則加倍self.windowsize並移動window:

```
#? Get window's ACK and complete in time -> SLIDE WINDOW
elif ack_num == self.windowsize and all_in_time:
    window_head = window_end
    self.windowsize*=2
    window_end = min(window_end + self.windowsize, packet_num)
    #? Send the window data
```

若在timer的時限內window中有packet沒收到ACK，則根據ack_rate有沒有小於self.expect_ackrate來決定要不要縮小self.windowsize(小於就大小減半)，並找出index最小的packet_index，將window的頭移動到那，並重新傳送self.windowsize個packets:

```
#? Did NOT get all ACK and Time is up for any one packet -> SLIDE WINDOW
elif not all_in_time:
    if min_noack_index != float('inf'):
        window_head = min_noack_index
    if((ack_num/self.windowsize) < self.expect_ackrate):
        self.windowsize = int(self.windowsize/2)
        if self.windowsize == 0:
            self.windowsize = 1
        # print("windowsize/2 because of actual ack_rate is lower than self.expect_ackrate!")
    window_end = min(packet_num, window_head + self.windowsize)
    #? Send the window data
```

這樣就達到了Congestion control 的效果(根據傳送情況調整送的packet數)

4. **If you use two streams to send data simultaneously from the client to the server or in the other direction, what will happen if one packet of a stream gets lost? Is the behavior of QUIC different from that of TCP? Why? 10%**

   因為ACK和Timer的機制，那個消失的stream會重傳，而QUIC允許其他stream繼續處理/傳送，而無需等待這個重傳的stream。

   這個現象和TCP不同，TCP雖然也有ACK和Timer的機制，但由於TCP重傳過程是有順序的，這意味著在接收到丟失的packets之前，接收方無法處理後續數據包。

   綜上所述，如果同時有兩個stream時，一個stream中若有packets丟失，對TCP將造成處理後續data的延遲，直到丟失的packets被重新傳輸和接收。相比之下，QUIC允許其他stream繼續處理而無需等待重傳，從而提供更好的整體性能和響應能力。