Project 1 Report

Yuxuan Zhang, Yuhang Wang

Email: YUZ276@pitt.edu, YUW199@pitt.edu

Phone: 412-979-8762

Contribution: Both members have the same contribution to the project.

Final design

Data Structures for packets and acks

```
struct upkt {
    int64_t     ts_sec;
    int32_t     ts_usec;
    uint32_t    seq;
    char        payload[BUF_SIZE];
    uint32_t    is_last;
 };

//structure for ack and nack
struct ack {
    uint32_t    cu_ack;
    uint32_t    nack_start;
    uint32_t    nack_end;
    uint32_t    is_last;
};
```

- The `is_last` of `struct upkt` has 2 possible values. `is_last == 0` means regular packets, `is_last == 1` means the packet is the last pacekt.
- The `is_last` of `struct ack` has 3 possible values. `is_last == 0` means regular packets, `is_last == 1` means that the ack is the last cumulative ack, telling the sender to terminate. `is_last == 2` means that the sender should be blocked and not sending messages. `is_last == 3` means that the sender can continue to send messages and it is not blocked.
- The `cu_ack` of the `struct ack` is the cumulative ack.
- I will go through `nack_start` and `nack_end` in details in the following section discussion about the receiver side.

Sender

Important Parameters

```
is_close = FALSE;
wait_exit = FALSE;
is_bootstrap = 0;
//int block = 0;
int first_one = 1;
int flag = 1;
int last_counter = 0;
pck_timeout.tv_sec = 0;
pck_timeout.tv_usec = 100;
timeout.tv_sec = 0;
timeout.tv_usec = 0;
```

Bootstrap stage

- At the start of the sending process, we will firstly send the a packet containing the dest_file name to the server.
- Then, the sender will do nothing but check for timeout of this first packet until it receive messages from the server.
- If the server send a packet with `is_last` as `2`, the sender will block itself for forever until the server tells the sender to go again. No worries about the messages of unblock from the server is lost, the server has its timeout system for the unblock message as well.
- Then, the sender knows that the real transmission shall begin and it will enter a bootstrap stage. It will send as many packets as the window can be filled.
- Then, it will begin to wait for ack.

Window

- Our window is an **array** of struct object.

```
struct upkt {
    int64_t  ts_sec;
    int32_t  ts_usec;
    uint32_t seq;
    uint32_t is_last;
    char     payload[BUF_SIZE];

 };

struct upkt* sender_window = (struct upkt*)malloc(sizeof(struct upkt) * window_size);
```

- We will take the result of packets' sequence number mod the size of the window.

```
sender_window[seq % window_size] = send_pkt;
```

If the window size is 5, and we have first 5 packets with sequence number from 1 to 5. Then, packet 1 to 4 will be stored at index 1 to index 4, and packegt 5 will be stored at index 0, since 5 mod 5 is 0. As the sequence number goes higher, for example, the sequence number 102 will be stored at index 2 since 102 mod 5 is 2.

How to slide the window

The left parameters

- Since our window is just a simulation of the real window on a fixed array.
- For example, a window of 5 containing packets number of `5, 6, 7, 8, 9.` Let's say that at this point , the cumulative ack is 4, and 5 is the smallest sequence number of unacked packets. We will then assign `left` variable as the index of this packets at any time. It will always be the index of the smallest sequence number of unacked packets.
- We only care about whether there is a free space to send new pacekts.
- Since our window starts from a full status. Please see the above bootstrap stage. We start from a full window, which means we can't send any new packets until the packets with the smallest sequence number is ackowledged.
- Therefore, in each iteration of the for loop, we will firstly attempt to listen from the receiver. And then we will go to line 7. We will always check the sequence number of the packets at the `left` index. We want to see whehter it is smaller than the cumulative ack. If so, we know that we have a free space to send new packets.
- Thus, we will send new packets, and store the new packets at `left` index, and update the `left` index to `left + 1` . If the `left` index is at the end of the array, we will reset left as 0.

Sliding the window

```
for (;;;){
  num = select();
  if (num > 0){
    //hear message from the sender
    doing things like update ack, resend packets according to the NACK
  }
  if(sender_window[left].seq <= cu_ack_seq){
     we will slide the window and send a new packet.
  }


}
```

Timeout

- We won't check for timeout if we are able to send new pacekts. In a word ,the window has a free space.
- Else, we will check the packet with `left` index at the window. We will always monitor for this timeout, if

the timeout has passed, we will resend it.

Receiver

This is the overall structure of the receiver process.

1. For each packet we receive, firstly **check** the sender. If the **sender** is not the client we are serving, **enqueue this sender** in our queue and send **block** message.
2. **Receive** packets from the correct sender, send corresponding **cumulative ACK and NACK**.
3. If we receive the packets with **no gap**, we directly **write** it to the file.
4. Otherwise, a **gap** is detected. We then **buffer** the packet with **inconsistent sequence**.
5. **Finally**, after finish writing, notify the **next client** in our queue that it is **unblocked**. If we **not receive** message from the **next client** for a period of time, the **timeout is triggered** and we will **resend** the unblock message.

Receiver window

Unlike the structure presented in the ppt, we only store packets which are needed to be buffered in our window. Upon packets with correct sequence number(no gap), we will directly deliever it.

Data structure

- Our window is an **array** of struct object.

```
struct pack_info {
    struct upkt packet;
    char status; // 'e' = empty, 'b' = buffer
};

window = (struct pack_info*)malloc(sizeof(struct pack_info) * window_size);
```

I **buffer** each packet with a character flag denoting the status of the packet. `b` means the packets at the index is buffered, and `e` means that there is no packets buffered at the index.

- We will take the result of packets' sequence number mod the size of the window.

```
int position = recvd_pkt.seq % window_size;
window[position].packet = recvd_pkt;
```

If the window size is 5, and we have first 5 packets with sequence number from 1 to 5. Then, packet 1 to 4 will be stored at index 1 to index 4, and packegt 5 will be stored at index 0, since 5 mod 5 is 0. As the sequence number goes higher, for example, the sequence number 102 will be stored at index 2 since 102 mod 5 is 2.

How to slide the window

```
for (;;;){
  num = select();
  if (num > 0){
    //hear message from the sender
    doing things like delivered, buffer message or send ack
  }

  uint32_t expect = cu_ack + 1;
  uint32_t seek = expect % window_size;
  while (window[seek].status == 'b'){
    // the next packet to be written is buffered at the window
    // we then deliver it and slide the window
    deliver it.
    window[seek].status = 'e'
  }

}
```

- Please see the above code. After processing the message sent by the sender, the program will then go to line 8.
- We will check and slide our window at this step.
- `expect = cu_ack + 1` means that the variable `expect` is always the next packet we are waiting to write in our file. We will use the `expect & window_size` to find the location or index whether it is going to be stored at or it is already be stored at.
- If at that index, the `(window[seek].status == 'b'`, then we know that we have the packet and all of the inforamtion stored in the array at that index, and we will take it our and deliver it to the file.
- Then, we will set the status of that index to empty, like `window[seek].status = 'e'`, even though the data is still stored at that index, but once the status is changed to `e`, we will be able to overwrite the data for future uses. Therefore, we don't need to erase the data. This is how we slide the window.
- For example, if at some point, we have packets starting from sequence 8 to 20 buffered at our window, but we did not receive packet 7 yet, so we can't deliver them. Then, at next iteration, we firstly hear from the sender and get the packet 7, we will deliver it right away and then go th line 10 to check packets in our buffer, we then find out that what we expect is packet 8 (since cumulative ack is just updated as 7). Therefore, within that while loop, we will be able to deliver all packets from sequence 8 to 20, since all of them have `status == 'b'.`

The algorithm of buffering data and sending NACK

1. What we need:

- In the class, we have the example of receiving packets in this sequence: 1,2,3,7,8,10
- In this circumstance, we want to send NACK of packet 4 to packet 6 upon receiving packet 7. Since we have

a gap between the previous sequence 3 and the current sequence 7. Then, we should be able to explicitly ask for retransmission of packet 4, 5, 6, and buffer packet 7 in our array, which is our window.

- Instead of put 4, 5, 6 in an array and send the array to sender, we only need to send two integers to indicate the whole restransmission. I will go through the details in a minute. For example, upon receiving 7, we will send `nack_start == 4` and `nack_end == 6`. In this way, the sender knows that it should retransmit all pakcets starting from sequence 4 to 6.
- Upon receiving 8, we have already send the NACK from 4 to 6 and buffer packet 7, then we shouldn't be wasting the internet resources and ask for retransmission again. All we need to do is buffer packet 8.
- Similarly, upon receiving packet 10, we only need to ask for retransmission of packet 9 and buffer packet 10.

2. Here is my implementation:

- `cu_ack` is only the sequence of packets delivered up to. However, `cu_already` means the sequence of all packets either is delivered or buffered or already asked for restransmit. For example, upon 1, 2, 3, 7. At sequence 7, we will have `cu_ack = 3` and `cu_already = 7`.
- Upon receving a packet, if the sequence number is equal to `cu_ack + 1`, we deliver it. Otherwise, if we haven't buffer the packet, we buffer it. Then, we check for the `cu_already`. If the sequnce number is greater than `cu_already`, we update the `cu_already` to this packet's sequence number. Then, we will send `nack_start` starting at `cu_already + 1`, since `cu_already` is the cumulative sequence number of all packets either is delivered, buffered or already ask for retransmit. And the `nack_end` will be the sequence number of this packet - 1.
- Continuing the above example, If we receive 8, then, 8 is not greater than `cu_already + 1`, because we have `cu_already` as 7. Thus, we don't send any NACK, we just buffer packet 8. Then, `cu_already` is updated as 8.
- Then, If we receive 10, 10 is greater than `cu_already + 1, which is 9`, then we know that something needs to be retransmitted, and the `nack_start = cu_already + 1`, `nack_end = recvd_pkt.seq - 1`. Thus, we have both `nack_start` and `nack_end` as 9, the sender knows that it only need to retransmit packet 9.

```
 uint32_t cu_ack, cu_already = 0;
if ((recvd_pkt.seq > cu_ack) && (window[position].status != 'b')){
  if (recvd_pkt.seq == cu_ack + 1){
     deliver it
  }else{
     buffer the packets
     if (recvd_pkt.seq > (cu_already + 1)){
         nack_start = cu_already + 1;
         nack_end = recvd_pkt.seq - 1;
         cu_already = recvd_pkt.seq;
    }
  }
```

Cumulative ack

We send cumulative ack upon every packet receipt. However, if the receive packets' sequence is less or equal to our current cumulative ack, we don't send anything, just to save internet resources.

Multiple client mangement system

- We stored each client in a FIFO queue structure.

```
struct node {
    char sbuf[NI_MAXSERV];
    struct sockaddr_storage from_addr;
    struct node* nextnode;
};

typedef struct node* node_t;

struct queue {
    node_t head;
    node_t tail;
    int count;
};
```

- We store their ip addresses and the port number.

implementation

```
ret = getnameinfo((struct sockaddr *) &from_addr, from_len, hbuf,
                                  sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST |
                                                                    NI_NUMERICSERV);
if (serve_first == 1){
   // it is a new client
  current_sbuf = sbuf;
}else if (strcmp(current_sbuf, sbuf) != 0){  // the client is not what we are serving
right now
   if (not in queue){   // we found a new clinet
     queue_enqueue(client_queue, this_packet); // enqueue the packets
   }
  send_ack; // tell the client it should be blocked
}else{
  serving the client that we should serve

}
```

- After finish the current client, we dequeue a client from the queue, and notify the client that it is unblocked right now. If we do **not receive** message from the **next client** for a period of time, the **timeout is triggered** and we will **resend** the unblock message.

How do receive and sender knows that it should exit

- sender has timeout for each packets.
- Since our server is meant to accept unlimited transfer request, it should not terminate.
- Therefore, sender will continuously send the last packet to the receiver until it receive the ack of the last packet.
- The receiver will close the file immediatly after receive the last packet and start serving the next client.
- Thus, the receiver may still receive packets from the last sender. At this circumstance, the receiver will send ack to the last sender to let it know that it can exit.
- I know that the exit messages from both side could be lost, but the sender has a timeout process and it is guaranteed to not exit until receive the last ack.

Other Details

Different Network Environment Handling

When running sender process, user should specify the desired network environment, namely local area network(LAN) and wide area network(WAN). Our program will use different window size for different network environments, we use 3000 packets for LAN and 40 packets for WAN.

First packet

In order to tell the receiver the file name desired on the receiver side (destination file name), our first packet is not the real content from the source file but the destination file name. We introduced a parameter first_one on the sender side to indicate if the sender needs to pack the first packet. Upon receiving the first packet on the receiver side, the receiver will use the payload which is the destination file name in the fopen function to generate file.

Performance results

In this section, we will be only listing results, further evaluation will be presented in the next section.

Local area evaluation

1. TCP Benchmark

   Transfering a 100-Megabyte file in LAN using t_ncp/t_rcv processes would get the following results as shown in Figure 1. As it is metioned in the project requirement, these results will be used as a benchmark to analyze the performance of our UDP file transfer system.

   The total bytes for transfered is slightly bigger than 100-Megabyte(100,000,013bytes), this is is an automatic generation result and there is nothing we can do about it. We will be using the same file in the following UDP experiments, so it does not affect our results.

Loss rate(%)	No	Total transfer time(sec)	Transfer rate(megabits/sec)
	1	25.0713	31.9089
	2	25.2478	31.6859
	3	25.3033	31.6164
	4	24.7037	32.3838
	5	25.3072	31.6115
0	average	25.12666	31.8413

Figure 1. Results of TCP in LAN

2. UDP transfer with different loss rate

   As required in the project, in a Locan Area Network environment, we run our UDP protocol to transfer the 100-Megabyte file under different loss rate conditions for 5 times each, the results are shown in Figure 3.

   Also as required, to better present the result of total transfer time among different loss rate, we draw Figure 2 to give a better view of the total transfer time on different loss rates.

Figure 2. Total transfer time of different loss rate

Loss rate(%)	No	Total transfer time(sec)	Transfer rate(megabits/sec)
	1	13.9794	57.2269
	2	15.2478	52.4667
	3	15.3033	52.2762
	4	14.7037	54.4081
	5	15.3072	52.263
0	average	14.90828	53.72818
	1	15.7042	50.9417
	2	15.1847	52.6847
	3	15.4269	51.8574
	4	15.177	52.7114
	5	14.8848	53.7462
1	average	15.27552	52.38828
	1	14.5426	55.0108
	2	14.8676	53.8084
	3	14.8237	53.9676
	4	14.9701	53.44
	5	15.2603	52.4236
5	average	14.89286	53.73008
	1	14.0128	57.0907
	2	13.7719	58.0892
	3	14.0675	56.8688
	4	14.5302	55.0578
	5	14.0042	57.1258
10	average	14.07732	56.84646
	1	13.9186	57.477
	2	13.9675	57.2758
	3	14.072	56.8505
	4	13.9977	57.1521
	5	13.9349	57.4098
20	average	13.97814	57.23304
	1	19.3121	41.4248
	2	18.7783	42.6024
	3	18.858	42.4222
	4	18.8427	42.4568
	5	19.0217	42.0572
30	average	18.96256	42.19268
	1	39.9134	20.0434
	2	39.8762	20.062
	3	39.9428	20.0286
	4	39.6337	20.1848
	5	39.8526	20.0739
0(with tcp)	average	39.84374	20.07854

Figure 3. Results of UDP under different loss rate in LAN

Wide area evaluation

1. TCP Benchmark

   Transfering a 100-Megabyte file in WAN using t_ncp/t_rcv processes would get the following results as shown in Figure 4. As it is metioned in the project requirement, these results will be used as a benchmark to analyze the performance of our UDP file transfer system.

   Our TCP has some packet lossing issues under WAN environment, normally it will loss up to 240 packets in sending the 100-Megabytes file which will take up tp 100000 packets. We are still working on solving this issue, but it is time to submit our results. We provide a test results of our TCP transfer here, please be aware that this is not the correct results.

Loss rate(%)	No	Total transfer time(sec)	Transfer rate(megabits/sec)
0(with tcp)	1	85.6296	9.3426

Figure 4. Results of TCP in WAN

2. UDP transfer with different loss rate

   As required in the project, in a Wide Area Network environment, we run our UDP protocol to transfer the 100-Megabyte file under different loss rate conditions for 5 times each, the results are shown in Figure 6.

   Also as required, to better present the result of total transfer time among different loss rate, we draw Figure 5 to give a better view of the total transfer time on different loss rates.

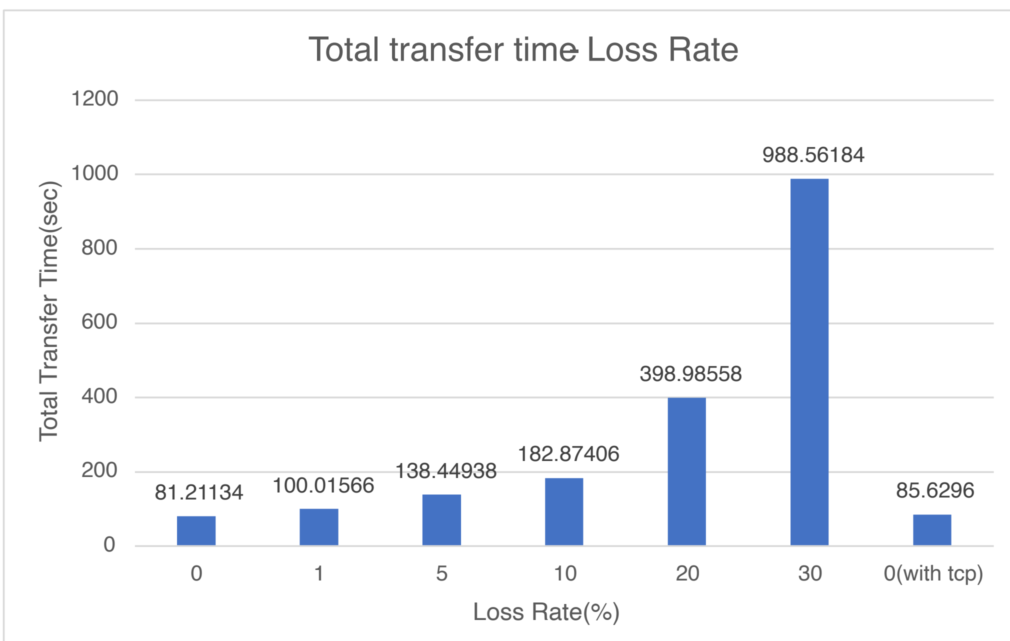

Figure 5. Total transfer time of different loss rate

Loss rate(%)	No	Total transfer time(sec)	Transfer rate(megabits/sec)
	1	81.2531	9.8458
	2	81.2467	9.8466
	3	81.1329	9.8604
	4	81.1849	9.854
	5	81.2391	9.8475
0	average	81.21134	9.85086
	1	100.3256	7.974
	2	100.1345	7.9893
	3	99.9123	8.007
	4	99.3022	8.0562
	5	100.4037	7.9678
1	average	100.01566	7.99886
	1	138.3252	5.7835
	2	138.8676	5.7608
	3	138.8237	5.7627
	4	137.9701	5.7983
	5	138.2603	5.7861
5	average	138.44938	5.77828
	1	182.9965	4.3717
	2	182.7719	4.377
	3	183.0675	4.3652
	4	182.5302	4.3828
	5	183.0042	4.3714
10	average	182.87406	4.37362
	1	398.9558	2.0052
	2	398.9675	2.0051
	3	399.072	2.0046
	4	398.9977	2.005
	5	398.9349	2.0053
20	average	398.98558	2.00504
	1	988.3085	0.8095
	2	988.7783	0.809
	3	988.858	0.809
	4	988.8427	0.809
	5	988.0217	0.8096
30	average	988.56184	0.80922
0(with tcp)	1	85.6296	9.3426

Figure 6. Results of UDP under different loss rate in LAN

Discussion of the performance results

UDP under LAN

As illustrated in Figure 3, our UDP protocol could maintain a transfer time of under 20 seconds even when the loss rate is up to 30%. We would like to mention that based on our test for TCP transfering, the average time is 25.12666 seconds. It is obvious that our UDP protocol has a better performance than TCP in transfering file among all the loss rate tested. We are exciting about this result!

However, in the last column of Figure 6, it is shown that when running at the same time with TCP, our UDP protocol would be strongly affected. We are guessing that this is because the principals defined when desinning TCP and UDP that when they are running at the same time, TCP will be the one who gets the bandwidth resources.

UDP under WAN

As shown in Figure 6, the total transfer time will strongly increase when loss rate increases under WAN environment.

Parameter tuning

Window size

We use dichotomy to find the best window size for our protocol under different network environments. We use 3000 in LAN and 40 in WAN.

Timeout

We use dichotomy to find the best timeout time for our protocol under different network environments. We use 750us in LAN and 40900us in WAN.

Port

We find it interesting that specific port also affects our protocol's performance, namely port 5500 performs better than port 5600, so we choose 5500 on the receiver side.