RDT communication program design

1. Waiting Agreement

1.1 Receiver protocol writing

Step 1 : Accept the RDT packet, receive the packet from the sockfd socket and put it into rdt _ pkt.

```
char rdt_pkt[RDT_PKT_LEN];
int pkt_len = recv(sockfd, rdt_pkt, RDT_PKT_LEN, 0);
```

Step 2 : Unpack the RDT packet, and unpack the data in rdt _ pkt into rdt _ data (data), pkt _ len (data length), seq _ num (data number), flag (packet type).

```
char rdt_data[RDT_DATA_LEN];
int seq_num, flag;
Int data_len = unpack_rdt_pkt(rdt_data, rdt_pkt, pkt_len,&seq_num,
&flag);
```

Step 3: Check whether the accepted packet is the expected packet. If it is, write it to the file. If not, do not accept it.

```
if (seq_num < exp_seq_num)
{
         printf("[Receiver]Packet #%d received, send ACK, already
received.\n", seq_num);
        }
else if (seq_num == exp_seq_num)
{
        fwrite(rdt_data, sizeof(char), data_len, fp);
        total_recv_byte += data_len;
        exp_seq_num++;
        printf("[Receiver]Packet #%d received, send ACK, now
exp_num is %d\n", seq_num, exp_seq_num);
        }
else {
        continue;
}
```

Step 4 : Encapsulate a new RDT packet, the ACK packet.

```
char reply_rdt_pkt[RDT_PKT_LEN];
int reply_pkt_len = pack_rdt_pkt(NULL, reply_rdt_pkt, 0, seq_num,
RDT_CTRL_ACK);
```

Step 5: Call UDP to send a new RDT packet

```
            udt_send(sockfd, reply_rdt_pkt, reply_pkt_len, 0);
```

Step 6: Determine if the end of the packet is reached, and jump out of the loop if the end is reached.

```
            udt_send(sockfd, replyif (flag == RDT_CTRL_END)
            {
                         break;
                    }
```

1.2 Sender side protocol writing

Step 1: Send RDT packets until the end of the file. If the end of the sending file has been read, set the packet type to RDT_CTRL_END, if it does not reach the bottom, set the packet type to RDT_CTRL_DATA.

```
      int counter = 1;
      char rdt_data[RDT_DATA_LEN];
      int flag, data_len;
      if (feof(fp))
      {
      flag = RDT_CTRL_END;
      data_len = 0;
      }
      else
      {
      flag = RDT_CTRL_DATA;
      data_len = fread(rdt_data, sizeof(char), RDT_DATA_LEN, fp);
      }
```

Step 2: Encapsulate RDT packets

```
      char rdt_pkt[RDT_PKT_LEN];
      int pkt_len = pack_rdt_pkt(rdt_data, rdt_pkt, data_len, seq_num, flag);
```

Step 3: Send RDT packet, and perform retransmission action if no ACK is received.

```
      while (1)
      {
              // step 2-1. Call unreliable data transfer to send a new RDT packet
      printf("[Sender]Packet #%d: %d bytes. Send count #%d\n", seq_num,
pkt_len, counter++);
        udt_send(sockfd, rdt_pkt, pkt_len, 0);
        // step 2-2. Wait until readable data is available for a file in the file
descriptor set, or until the timeout limit is reached :poll()
        struct pollfd pollfd = {sockfd, POLLIN};
        int state = poll(&pollfd, 1, RDT_TIME_OUT);
        if (state == -1)
        {
        printf("Socket error.\n");
            eturn 1;
        }
        else if (state == 0)
        {
        printf("Timeout, resend....\n");
        continue;
        }
        else if (state == 1)
        {
        char reply_rdt_pkt[RDT_PKT_LEN];
        int reply_pkt_len = recv(sockfd, reply_rdt_pkt, RDT_PKT_LEN, 0);
        int reply_seq, reply_flag;
        unpack_rdt_pkt(NULL, reply_rdt_pkt, reply_pkt_len, &reply_seq,
&reply_flag);
        if (reply_seq == seq_num && reply_flag == RDT_CTRL_ACK) {
        printf("[Sender]Receive ACK for packet #%d\n", seq_num);
        break;
        }
        }
```

Step 4: If the send is successful, update seq_num and total_send_byte

```
        seq_num++;
        total_send_byte += data_len;
        if (flag == RDT_CTRL_END)
        {
        break;
        }
```

1.3 Experimental results test

Type cat/dev/urandom |base64| head -n 20000 >test in the terminal interface to generate a random data of 20000 lines into the file test. type . /rdt_stopwait_receiver testrecv &&md5sum test testrecv. Call the receiver function to accept the data into the testrecv file and hash it to extract the file data and compare the similarities and differences between the two. Type in .

/rdt_stopwait_sender test , send the test file.

2、Return N protocol

2.1 Writing the code on the sender side
GBN sender events and actions:
-Calls from the upper layer: When the upper layer calls rdt_send(), the sender checks if the window is full. If it is not full, it generates a group to send it and updates the variables accordingly; if it is full, it just returns the data to the upper layer and implicitly indicates that the window is full. (In practical implementations, the sender can cache this data or use a synchronization mechanism that sets up a semaphore and allows the upper layer to call only when the window is full).
-receiving an ACK(n), which in the GBN protocol uses a cumulative acknowledgement, indicating that the receiver has correctly i.e. received all previous packets including those within n with serial number n.
-Timeout event. If a timeout occurs, the sender retransmits all packets that have been sent but not acknowledged.

Step 1: If the packet at the left end of the sliding window has received an ACK, slide the sliding window to the next location where the ACK packet was not received

```
    if (ptr_pkt_left->state == RDT_PKT_ST_ACKED) //有 ACK 包到达
    {
    int max_acked = send_window.left;
       while(max_acked<send_window.right&&(send_window.rdt_pkts[max
_acked%send_window.len].state==RDT_PKT_ST_ACKED))
    max_acked++;
    send_window.left = max_acked;
    }
```

Step 2: Load the new packet into the vacant position at the right end of the sliding window

```
    int pkt_to_send = 0; // Number of packets to be sent
    for (int i = send_window.right; i < send_window.left + send_window.len;
i++)
      {
    if (feof(fp)) //Check if the end of the send file has been reached
    break;
    char data_buf[RDT_DATA_LEN];
    int data_len = fread(data_buf, sizeof(char), RDT_DATA_LEN, fp);
//Read the data in the send file
    StatePkt *ptr_pkt = &send_window.rdt_pkts[i % send_window.len];
    int rdt_pkt_len=pack_rdt_pkt(data_buf,ptr_pkt->rdt_pkt, data_len, i, RDT_CTRL_DATA); //Encapsulation as RDT packets
    //Initialize new RDT packet state
    ptr_pkt->pkt_seq = i;
    ptr_pkt->pkt_len = rdt_pkt_len;
    ptr_pkt->state = RDT_PKT_ST_INIT;
    pkt_to_send++;
    }
    send_window.right += pkt_to_send;
```

2.2 Write the code on the receiving end

GBN receiver events and actions.
In GBN, if a packet with sequence number n is received correctly and in order (i.e., the last data delivered to the upper layer to the upper layer was a packet with sequence number n-1), the receiver sends an ACK for packet n and delivers the data in that packet to the upper layer. In other cases, the receiver discards the packet and retransmits an ACK for the most recently received packet in order.

```
    //Receive RDT packets until all data has been received
        while (1) {
            // step 1. Receiving RDT packets
            char rdt_pkt[RDT_PKT_LEN];
            int pkt_len = recv(sockfd, rdt_pkt, RDT_PKT_LEN, 0);

            // step 2. Decapsulating RDT packets
            char rdt_data[RDT_DATA_LEN];
            int seq_num, flag;
            int data_len = unpack_rdt_pkt(rdt_data, rdt_pkt, pkt_len, &seq_num, &flag);

            //step 3.Check if this packet is the expected packet : seq_num==exp_seq_num
            if (seq_num < exp_seq_num) {
                printf("[Receiver]Packet #%d received, send ACK, already received.\n", seq_num);
            } else if (seq_num == exp_seq_num) {
                fwrite(rdt_data, sizeof(char), data_len, fp);
                total_recv_byte += data_len;
                exp_seq_num++;
                printf("[Receiver]Packet #%d received, send ACK, now exp_num is %d\n", seq_num, exp_seq_num);
            } else {
                continue;
            }

            // step 4. Encapsulate a new RDT packet (ACK packet)
           char reply_pkt_buf[RDT_PKT_LEN];
            int reply_pkt_len = pack_rdt_pkt(NULL, reply_pkt_buf, 0,
seq_num, RDT_CTRL_ACK);

            // step 5.Call Unreliable Data Transfer to send a new RDT
packet (ACK packet)
            rdt_send(sockfd, reply_pkt_buf, reply_pkt_len, 0);
            if (flag == RDT_CTRL_END) {
                break;
            }
        }
```

2.3 Program result test
Type cat/dev/urandom |base64| head -n 20000 >test in the terminal interface to generate a random data of 20000 lines to the file test. type . /rdt_gbn_receiver testrecv &&md5sum test testrecv. Call the receiver function to accept the data into the testrecv file, and do a hash to extract the file data and compare the similarities and differences between the two. Type in . /rdt_gbn_sender test , send the test file.

3、Select the retransmission protocol

3.1 Writing the code on the sender side
Select events and actions for retransmitting the sender
- Data is received from the upper layer. When data is received from the upper layer, the SR sender checks for the next available serial number for that packet. If the serial number is within the sender's window, the data is packetized and sent; otherwise it is either cached or returned to the upper layer for later transmission, just as in GBN.
- Timeout. Timers are used to prevent lost packets. However, each packet must now have its own logical timer, since only one packet can be sent after a timeout occurs. A single hardware timer can be used to emulate the operation of multiple logical timers.
- If an ACK is received, the SR sender marks the acknowledged packet as received if the packet's serial number is in the window. If the number of the packet is equal to send_dase, the window is moved forward to the unacknowledged packet with the smallest number. If the window is moved and there are unsent packets whose serial numbers fall within the window, these packets are sent.

```
else{
int i;
for( i = sending_window.send_left; i< sending_window.send_right; i++)
{
STATE_PKT *ptr_pkt = &sending_window.rdt_pkts[i % sending_window.win_len];
if(time_out(time_now,ptr_pkt->send_time) && ptr_pkt->state == RDT_PKT_ST_SENT)
{
printf("[mainthread]Packet #%d timeout.\n",ptr_pkt->pkt_seq);
ptr_pkt->state = RDT_PKT_ST_TMOUT;
pkt_to_send ++;

}
}
}
```

3.2 Write the code on the receiving end

Select events and actions for retransmitting receivers
-The packet with serial number within [rcv_base,rcv_base+N-1] is received correctly. In this case, the received packet falls within the receiver's window and a select ACK is sent back to the sender. If the packet has not been received before, the packet is cached. -- If the group's serial number is equal to the base serial number of the receive window, the group is delivered to the upper layer, along with the previously cached group with consecutive serial numbers. The receive window then delivers these groups up by the number of the forward-moving group.
-Packets with a serial number within [rcv_base-N,rcv_base-1] are received correctly. In this case, an ACK must be generated, even if the packet was previously acknowledged by the receiver.
-Other cases. The packet is ignored.

```
//step 1. Receiving RDT packets : recvfrom()
pkt_len = recvfrom(sock_fd, rdt_pkt, RDT_PKT_LEN,0,(struct sockaddr *) &client_addr,&sin_len);
//step 2. Decapsulating RDT packets : unpack_rdt_pkt()
data_len = unpack_rdt_pkt(rdt_data,rdt_pkt,pkt_len,&seq_num,&flag);
printf("[Window]=[%d,%d),
receive %d\n",recv_window.recv_left,recv_window.recv_right,seq_num);
//step 3. Check if this packet is the expected packet
if(seq_num <= recv_window.recv_right)
{//step 4. Encapsulate a new RDT packet (ACK packet)
reply_pkt_len =
pack_rdt_pkt(NULL,reply_pkt_buf,0,seq_num,RDT_CTRL_ACK);
    //step 5. Call Unreliable Data Transfer to send a new RDT packet (ACK
packet): udt_sendto()
    udt_sendto(sock_fd,reply_pkt_buf,reply_pkt_len,0,(struct        sockaddr
*)&client_addr,sin_len);
    if(seq_num >=recv_window.recv_left)
    {
    recv_window.rdt_pkts[seq_num    %    recv_window.win_len].state    =
RDT_PKT_ACKED;
    recv_window.rdt_pkts[seq_num    %    recv_window.win_len].pkt_len    =
pkt_len;
   memcpy(recv_window.rdt_pkts[seq_num%recv_window.win_len].rdt_pkt,
rdt_data,data_len);
    total_recv_byte += slide_window(&recv_window,fp,data_len);
    }
    }
    else{
printf("Left   state=   %d\n",recv_window.rdt_pkts[recv_window.recv_left   %
recv_window.win_len].state);
    }
    if(flag == RDT_CTRL_END)
    {
    break;
    }
```

Summary

Reflection Questions

The underlying layer of RDT is UDP, why can the program use recv/send instead of recvfrom/sendto to send and receive data?

Solution: The sockfd socket send/receive port has been binded in the net.c function, and the connect function is used to connect two hosts before calling send and recv, and the send() and recv() functions are encapsulated afterwards, which can send and receive data instead of sendto and recvfrom.

How is the timeout retransmission implemented in the stop and wait sender program?

Solution: Use the poll() function to listen for events, and define the maximum wait time for the parameter function in this function. Set the poll() function to return the state value. On success, poll() returns the number of file descriptors in the structure whose revents field is not 0; if no event occurs before the timeout and no ACK response is received, poll() returns 0; on failure, poll() returns -1, and set errno to one of the following values: EBADF(one or invalid file descriptor specified in more than one structure); EFAULT (fds pointer points to an address that is outside the process address space); EINTR (a signal is generated before the requested event and the call can be re-initiated); EINVAL (nfds parameter exceeds PLIMIT_NOFILE value); ENOMEM (insufficient memory available to complete the request), Performs a retransmission operation.

How to ensure that both sides end the communication correctly when there are sending and receiving failures (refer to the two armies against each other problem)?

Solution: Conclusion of the two-army problem in computer networks: This goes on in an infinite loop, and neither blue army can ever be sure that the last message it sent was received by the other. There is no protocol where the blues can win 100% of the time. Use the above problem to illustrate the main point of the "release connection" problem in network transmission: it is not possible to stop communication by confirming that both sides have received an acknowledgement of the other side's message; there is no such thing as an acknowledgement of an acknowledgement. In practice, when two computers interconnected through the network release the connection (corresponding to the issue of the two armies to initiate the attack), the connection is usually released after Party B receives an answer message from the other side confirming that it is not replying (using a three-step handshake protocol). A timer is set here, and when the time exceeds without receiving an acknowledgement message, this message is sent again, until the acknowledgement message is received; finally, the infinite loop is broken and the acknowledgement message is not confirmed so that it is processed. The protocol is not entirely faultless, but usually as well as sufficient, i.e., the use of three handshakes plus a timer to release the connection is successful in the vast majority of cases.

Why must the window size be less than or equal to half the size of the sequence number space in the selective retransmission protocol?

Solution: If the ACK packet corresponding to the leftmost packet of the window is lost during transmission, the sender needs to retransmit the leftmost packet of the window. If the window size is larger than half of the sequence number at this time, the right side of the window must contain the packet with the same serial number as the leftmost, and the receiver cannot distinguish the marked packet from the retransmitted packet on the left side of the window or the newly sent packet on the right side of the window when sending.

My Takeaways and Summary

The window size of the sender is determined by the receiver, with the purpose of controlling the sending speed so that the receiver's cache is not large enough to cause overflow, and controlling the traffic can also avoid network congestion. The theory of the sliding window protocol has different applications in practice. The first is the stop-wait protocol, when both the receiver's window and the sender's window size are 1. The sender then naturally sends only one at a time, and must wait for an ACK of this packet before sending the next one. Although it is less efficient and has significantly lower bandwidth utilization, it is nevertheless applicable in poor network environments or when the bandwidth itself is very low. The stop-and-wait protocol, although simple to implement and better suited to poor network environments, is clearly too inefficient. This is where the sliding window protocol really comes in, where the window size is n and the receiver's window is still 1. The benefit of the backoff n protocol is undoubtedly increased efficiency, but if the network is bad, it can lead to a lot of data retransmissions, which is not as good as the stop-and-wait protocol above, which is actually quite common, as can be seen in TCP congestion control. Another problem with the backoff n protocol is that when an erroneous frame appears, all frames after that frame are always retransmitted, no doubt further worsening the network conditions when the network is not very good. The retransmission protocol is used to solve this problem. The principle is simple: the receiver always caches all the received frames, and when an error occurs, it will only ask to retransmit this one, and only when all the frames after a certain number are received correctly, it will be submitted to the higher-level application. The disadvantage of the retransmission protocol is that more caches are needed on the receiving side.
The reason why the packet is not received after a while at the receiver side has occurred in the experiment: the receiver side replies with an ACK even if it receives a packet with a duplicate serial number, because the ACK packet replied by the receiver side may be lost.
It is impossible to guarantee that the message is sent successfully and both sides confirm that the message is received on an unreliable channel. That is, it is not possible to stop communication by confirming that both sides have been acknowledged by each other's messages, and there is no such thing as an acknowledgement of an acknowledgement. So we can send a message and not wait for the confirmation message from the other side all the time, but set a timer, and when the time exceeds without receiving the confirmation message, send this message again until the confirmation message is received; finally, break the infinite loop and do not acknowledge the confirmation message.