Lab1 RDT

Design and Implementation

I finished this lab with Selective Repeat strategy. Now I'd like to explain my design and implementation strategies in detail.

Sender Part

Packet Design

```
|<-  2 byte  ->|<-  4 byte  ->|<-  1 byte  ->|<-  1 byte  ->|<-  the rest ->|
|   checksum   |  packet seq  | is spilt msg | payload size |<-  payload  ->|
```

The header includes the checksum number, the packet sequence number, one byte indicating whether a packet belongs to a split message and the payload size. It takes up 8 bytes in total. The rest 120 bytes are used for the payload.

Get message From Upper Layer

When a message is passed from the upper layer at the sender, put the message in the buffer. Once the timer is not working, restart the timer and try to fill the window with the messages in the buffer.

When talking about filling the window, we maintain a variable called `msg_cursor` which is initialized to 0 and always points to the first unsent byte in the message. If `msg.size - msg_cursor > max_payload_size`, which mean just one packet can't contain the whole message, we set the `is_spilt_msg` in the packet header as true to show that this packet belongs to a split message , otherwise false.

For the first packet of a split message, we place the message size at the front of its payload.

After filling the window, we can send packets now. We maintain an array named `sack` to show whether the sender has got a packet's ack. If so, we don't need to send it again, because the receiver has saved it in its own window.

Get Ack From Lower Layer

When receiving an ack, check the checksum at first. If something goes wrong according to the different checksums, discard the packet. If the packet is not corrupted, set the packet as received-ack in the `sack` array.

We maintain a variable called `expect_ack`, which shows the sequence number of the expected ack(in other words, the packet with smallest sequence number hasn't got ack now). Once the ack number we received is equal to the `expect_ack`, we can move the window backwards to discard the packets which have received their acks because we won't need to resend them any more, until we meet a packet which hasn't received its ack. Then we can refill the window now.

If all the packets in the window get acks according to the `sack` array, we can stop the timer to show that.

Once Timeout

We use `sack` to find the packets that hasn't got acks, and resend them.

Receiver Part

Packet Design

```
        |<-  2 byte  ->|<-  4 byte  ->|<-   the rest  ->|
        |   checksum   |    ack seq   |<-   nothing   ->|
```

The packet is consist of a checksum number and an ack sequence number.

Receive Packets From Lower Layer

When receiving a packet, check the checksum at first. If something goes wrong according to the different checksums, discard the packet.

We maintain an array `acks` to show whether a packet is received. Once the receiver receive a packet, we read the packet sequence number, send an ack and mark the packet as received in the `acks`. If it is the first time to receive this packet, we put it into the `rev_window`.

We maintain a variable named `ack_seq` which means the sequence number of ack(it is also the next packet sequence number we expect to receive **in order**). Once the packet sequence number we received is equal to `ack_seq` , we can move the window backwards to handle all the following packets has being saved to get the messages.

When handling an packet for a new message, first check the `is_spilt_msg` in the header. If it is true, read the message size from the first 4 bytes of the payload. Then use the message size to check whether the whole message is received. If so, we get to know that the next packet belongs to a new message. We maintain a variable called `rev_msg_cursor` which is initialized to 0 and always points to the first unreceived byte in the message. It works well for collecting split message with several packets.

Result

Formal Testing

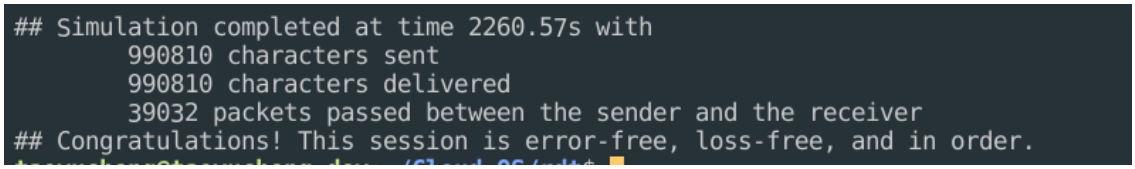
- `rdt_sim 1000 0.1 100 0.02 0.02 0.02 0`

```
## Simulation completed at time 1000.63s with
        991501 characters sent
        991501 characters delivered
        29175 packets passed between the sender and the receiver
## Congratulations! This session is error-free, loss-free, and in order.
```

Other Testing with Reference Value

- `rdt_sim 1000 0.1 100 0.15 0.15 0.15 0`

  Mine:

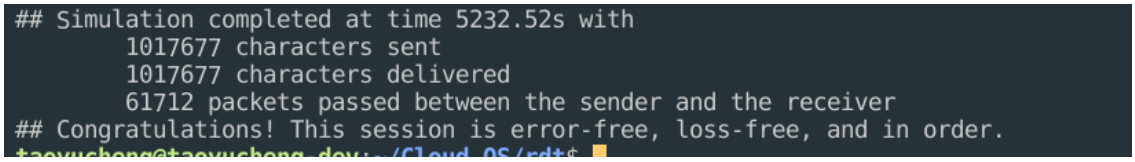
  

  Reference Value:

  ```
  ## Simulation completed at time 1656.02s with
     983959 characters sent
     983959 characters delivered
     38066 packets passed between the sender and the receiver
  ## Congratulations! This session is error-free, loss-free, and in order.
  ```

- `rdt_sim 1000 0.1 100 0.3 0.3 0.3 0`

  Mine:

  Reference Value:

  ```
  ## Simulation completed at time 4321.40s with
     984465 characters sent
     984465 characters delivered
     59417 packets passed between the sender and the receiver
  ## Congratulations! This session is error-free, loss-free, and in order.
  ```

Other file changes

In addition to `rdt_sender.cc` and `rdt_receiver.cc`, I also created files `rdt_protocol.h` and `rdt_protocol.cc` to set some protocols for my RDT implementation, such as the checksum function, window size and son on. And I also modified the makefile to make these two files work.

519021910594

陶昱丞

taoyucheng@sjtu.edu.cn