<u>Project 2 - Real-Time Transfer Protocol</u>

Ashwini Ananda
Jayati Halder Jui

1. Message Types: We have three types of messages; Data messages (Stream Packet and RT Packet) and Acknowledgements (Handshake and Nacks). We have three data structures to represent these:

struct stream_pkt: (This packet structure is used by udp_stream and udp_stream_rcv)

Data Type	Field	Comment
Int	Sequence No.	Sequence Number of packets
Int 32_t	Tv_sec	The timestamp sec
Int 32_t	Tv_usec	The timestamp micro sec
Char* [1300]	Data_buf	Data length

Struct rt_packet: (This packet structure is used by the protocol, to ease out time computations)

Data Type	Field	Comment
Int	Sequence No.	Sequence Number of packets
Timeval	ts	The timestamp
Char* [1300]	Data_buf	Data length

struct ack_pkt: (This is used for the handshake packets and nacks)

Data Type	Field	Comment
Int	ACK No	Acknowledgement number based on sequence no.
Struct timeval	Ts	Timestamp

1. Data Structure for Sender and Receiver:

Determining Window Size:

Assuming Bitrate 20Mbps, 1400-byte packets, and a maximum latency window of 1 second,

We will set Window size = bandwidth delay product

Ceiling (20 * 1000000) / (8 * 1400) = 1786

- The Sender and Receiver use ARRAY (packet*) of size 1786 to store the packets. The sender receives every incoming packets and place it in the array at index seq_no % 1786.

- The receiver receives every incoming packets except the duplicate ones and place it at index seq_no % 1786.

2. Design

- Sender Workflow:
    1. The server (server) initiates connection with a client through handshake
    2. The server receives input stream at the input stream port and send streams to the client port in real time and store them in buffer
    3. If the server get's another request from a client while processing previous request, it rejects the incoming client
    4. If the server receives NACK from the client, it checks the delivery timestamp of the NACK, and resends only if the sending time is still less than the delivery timestamps.
- Receiver Workflow:

    1. The client initiates connection with the server. If the server rejects the connection, the client exits

    2. The client receives incoming stream from the server. It then updates the timestamp for every packet as (receive_time + latency_window) and store them in buffer.

    3. If the client get's something out of sequence, the client sends NACKs to the server for the lost packets.

    4. It adjusts it's timeout based on the minimum(next_delivery_time, rtt) at every iteration and listens to the server socket for incoming streams. It updates the next deliverable packet sequence accordingly.

    5. At a timeout event, the client determines whether the time out event is for delivery. In case of a delivery event, the client delivers the next deliverable packet to the application port. Otherwise, the client sends nacks to the server in an attempt to recover lost packets.

6. To maintain synchronization, for each packet that the client gets as a response to a NACK, the client updates the delivery timestamp for these packets with the delivery timestamp with the next available packet in the buffer.

Some design insights:

1. Handshake: The server and client does a three way handshake. The client sends an ACK packet to the server with arbitrary large negative sequence.
    a. The server replies to the client for every negative ACK with the same negative ACK and timestamp.
    b. The client matches the first negative ACK it got back from the server, calculates the RTT based on receive_time – send_time. It accepts the connection and sends NACK for the first sequence.
    c. As soon as the client get's a NACK for the first sequence, it accepts the connection and proceed to deliver data streams to client.
2. Updating Client Timeout:
    a. As discussed above, the client updates it's timeout based on the minimum value of RTT and Next_Delivery_Time. Thus the server ensures that, it attempts to recover the lost packets until the delivery window is not passed.
    b. In the nack packets, the client set's the timesteps as delivery_time for that packet so that the server can determine whether the packet is still worth sending. In out local implementation, we implemented a checking for the client where it attempts to determine the deliverability of a packet based on the delivery_time – 2 * min_one_way_delay. It however didn't work in Geni because the Geni server has a positive clock_skew.
3. Delivering Packets:
    a. The client attempts to deliver packets in sequence. If a packet is not available at the delivery window, it simply moves forward to the next packet.