Birju Patel
Alex Kwiatkowski
Dr. Amy Babay
October 22, 2022

Project 2: Submission Document

Final Design, Packet Types

We created a UPKT packet type to send packets between rt_srv and rt_rcv. We utilized the provided STREAM_PKT packet type to send data between rt_rcv and udp_stream_rcv and rt_srv and udp_stream.

UPKT:

- Flags
  - INIT - Used to initiate a new connection between the client and server. Used to calculate the initial round trip time and time difference between the client and server.
  - LIVE - Used by the client to indicate that it is ready to receive data from the server. It is also used to periodically recalculate round trip time and clock drift in the extra credit program.
  - NACK - Used by the client to request a retransmission of a packet. It is also used by the server to deny a new connection request.
  - If no flag is set, the packet contains data that should be sent to the application.
- Timestamp - The time when the packet is sent.
- Sequence Number - Used to track which data packets are missing and to request retransmissions.
- Payload Size - The amount of data included in the payload
- Payload - In a data packet, this contains a stream_packet, which contains the data we get from udp_stream and send to udp_stream_rcv.

Final Design, Data Structures:

In rt_srv, we maintain an array called pkt_buf. We assume we send at a maximum rate of 20 Mbps, the client's latency window is capped at 1 second, and each packet contains 1400 bytes. So the maximum packets we need to buffer is given as 20Mbps * 1000000 bits / (8 bits/byte * 1400 bytes/packet) = 1786 packets ≈ 1800 packets = WINDOW_SIZE. When a packet sequence number x is sent, it is placed in position x % WINDOW_SIZE in pkt_buf. We look here when a retransmission request is made for a packet.

In rt_rcv, we maintain 2 linked queues, latency_buffer and missing_pkts. We keep packets that are received and waiting to be delivered to the application in the latency_buffer. Packets are removed from the front of the latency_buffer and delivered to the application. When we detect a gap in the transmission, we immediately send NACKs to the server to request retransmission. We take the sequence numbers of the lost packets and place them in the missing_pkt list. When those gaps are filled, we remove them from the list. Every so often, we iterate over the entire list and resend NACKs for all the missing packets.

Final Design, Server Side

The server listens on two ports, the app port and the client port. On the app port, it expects a constant stream of data from a udp_stream process. On the client port, it expects control messages from its client.

The server process can only handle 1 connection in its lifetime. When rt_srv starts, it is initially in the FREE state. If it receives an INIT packet from a client, it saves that client's IP address and port, transitions to the WAITING state, and echoes an INIT packet back to the client. If it receives any other packet type, it disregards it.

When the server is in the WAITING state, it is waiting for the client to signal that it is ready to start receiving data. From now on, if the server receives an INIT from another client, it rejects it, responding with a NACK. Also from now on, including in the BUSY state, every time the server gets a LIVE packet from its client, it will respond by sending another LIVE packet with the same sequence number and a current timestamp. When the server receives its first LIVE packet, it transitions to the BUSY state.

In the BUSY state, the server will take any data it receives from the udp_stream and will forward it to the client. Each packet the server sends is assigned a new sequence number and is buffered in pkt_list. When the server receives a NACK, it will look at the sequence number and search in the pkt_list for the corresponding packet, verify that it is the requested packet, and send it to the client. It will now stay in the BUSY state forever.

When the statistics timer goes off, we calculate and print summary statistics for the run.

Final Design, Client Side

The client goes through two stages, the initialization phase and the transmission phase. The receiver sends messages to and expects data from the given server IP and port. It forwards data to the given application port.

In the initialization phase, the client sends INIT packets to the server at regular intervals. When it receives an INIT in response, it calculates the round trip time and base delta using the formulas discussed in class. If a NACK is received in response, the receiver simply exits. For this design, we assume that after this, neither value changes. We account for changes in RTT and clock drift in our extra credit design. When this completes, we leave the initialization phase and enter the transmission phase.

The transmission phase begins by sending LIVE packets; these are sent at regular intervals, see the events section below, in case the first is dropped and allowing for recalculations in the extra credit. When a LIVE packet reaches the server in the WAITING state, it will begin streaming data packets as they become available from the app. From there, 6 possible events may occur. Not all events may be active at the same time. Most timeouts are not on unless certain conditions are met. All 6 events are monitored in a single select statement, which is nested inside the main control loop. The following is a list of the events at a high level:

- Received a packet from the server
    - LIVE packet: In this design, nothing is done. In the extra credit design, we use this to recalculate our RTT and base delta.
    - INIT or NACK packet: This packet is spurious, so we ignore it.
    - Data packet: This data packet has a sequence number that is equal to the expected sequence number (the normal case), higher than the expected number (indicating a gap), or less than expected (indicating that it is a retransmitted packet, we assume no packet reordering). Each case is handled differently.
- LIVE timeout: It is time to send a LIVE packet to the server.
- Statistics report timeout: It is time to display summary statistics for the current run.
- Deliver timeout: It is time to deliver the next packet from the latency_buffer to the application.
- NACK timeout: It is time to resend NACKs for every packet that is missing, as recorded in the missing_pkts list.

Consider that we receive a data packet. If this is the first data packet we have ever received, we begin the statistics timer and record this time as the beginning of the stream. We also check the size of the latency_buffer. All packets are removed from the latency_buffer at deliver_time = pkt.timestamp + base_delta + latency_window. If the latency_buffer is empty, we restart the delivery timer to trigger at this time.

We now check the sequence number. If the sequence number is the expected sequence number (highest number received + 1), we immediately put it in the back of the latency_buffer. If it is lower than expected, we assume this is the server's response to a previous retransmission request. We search the latency_buffer from front to back and put it in the proper place. We also search the missing_pkts list and remove this packet, as it is no longer missing. If it is higher than expected, there must be a gap in the transmission (gap = current - highest received). We create a NACK for every value in the gap. We send these NACKs now, and we add the values to the missing_pkts list. Note that we resend NACKs for every missing packet every 1.25 RTTs. If the missing_pkts list was previously empty, we begin this NACK timer.

When the statistics timeout occurs, we calculate and print summary statistics for the run so far. We then reset the statistics timeout. When the LIVE timeout occurs, we send a LIVE packet to the server, and then reset the LIVE timer.

When the delivery timeout occurs, it is time to send the next packet in the latency_buffer to the application. We remove the packet at the front of the queue and send it to the application. We then look at the new head of the queue and record its timestamp and sequence number. Using the formula deliver_time = pkt.timestamp + base_delta + latency_window, we reset the delivery timeout for the next packet. This packet is the next packet to be delivered to the application. It is possible that between the previous packet and this next packet, there was a gap of packets that are still recorded in missing_pkts and were not able to be delivered in time. At this point, it is too late for those packets, so we give up on them. We search through missing_pkts and remove any packets that have a smaller sequence number than the next packet.

When the NACK timeout occurs, it is time to resend NACKs for all missing packets. We loop through the entire missing_pkts list and send NACKs for them all. We then reset the NACK timeout to occur in another 1.25 RTTs.

Usage Information

The flag PRINT_DEBUG is present in both rt_srv and rt_rcv. To see debug messages, set this flag to 1 and recompile the code. A debug message will be printed out every time either program sends or receives a packet. Try this while sending a small number of packets from udp_stream to verify it is working as expected. To turn debug messages off, set the flag to 0.

The flag REALTIME_PROTOCOL in rt_rcv is used to implement extra credit 2. If this flag is set to 0, instead of using the protocol above, rt_rcv will use a simpler protocol. It will simply forward any data received directly to the application, and will not send retransmission requests. Brief summary statistics will also be printed. This flag should be set to 1 to use our protocol.

Extra Credit Opportunity 1 and 2

We modified our rt_rcv.c to only use UDP and circumvent our recovery protocol. We then streamed a video from the remote node to our computer at several loss rates and with our protocol both on and off. These videos are included in our submission.

Extra Credit Opportunity 3 Design (rt_srv_ec and rt_rcv_ec)

As explained above, we can use the LIVE packet mechanism to recalculate RTT and base_delta for the receiver to account for the shifting clocks for both the receiver and server and the shifting round trip time. To account for changes in RTT, the receiver might send another LIVE packet before the previous one was received, and our current protocol of just checking that the sequence number of the LIVE ACK matched the receiver's current live sequence number -1 would not work. Thus, the receiver now keeps track of the previous NUM_LIVE_TO_KEEP live packets and can use these to update the RTT and base_delta accordingly as opposed to only remembering the last sent LIVE packet. In order to update RTT, we first calculate the newly found RTT by subtracting when we sent the corresponding LIVE to when we received the server's response and then use a weighted average of the current RTT (40% weight) and the newly found RTT (60% weight). The same is done with base_delta with its corresponding calculation. Compared to the original protocol, these live packets are sent at a more frequent rate to allow for better estimates of clock drift and RTT.In order to test that this would allow our protocol to work dynamically, we implemented a mechanism for emulating clock drift on both the receiver and server and a mechanism for emulating changes in RTT on the server.

Starting with the clock drift mechanism, at the start of the program, a coin is flipped which decides whether that machine's clock is (artificially) fast or slow. Drift only takes effect once streaming has started. Once streaming has started, every 1 second for both the receiver and server, the drift will be incremented by a value between 0 us and MAX_DRIFT_US. Whenever the time is needed for an interaction between the server and receiver or for when the receiver needs to deliver, the time of day is retrieved and then adjusted (either incremented or decremented depending on the initial coin flip) by the current delay. For the purpose of printing statistics, the normal time is used.

To emulate delay, whenever the server is about to send something, either data/retransmission or live response, to the receiver, it will sleep according to the current delay. To implement this, since the time is calculated whenever the server is preparing a message, we just include the sleep after the drift time is calculated. Similar to drift, delay only takes effect once streaming has started. RTT delay will be in the range of 0 us to MAX_RTT_DELAY_US, and it will randomly change on an interval from 0 us to MAX_DELAY_TIMEOUT_US. Unlike drift, the new delay is not added to the previous delay after each timeout. The randomness in the delay timeouts better emulates the changes in available bandwidth, congestion, outages, etc.

Similar to PRINT_DEBUG, setting PRINT_DRIFT_AND_DELAY to 1 will display when delay has changed and what drift and delay have changed to. PRINT_DEBUG will also print both the actual time and the calculated drift time. If you wish to force the clock to always be slow or fast as opposed to random, the flag SET_CLOCK_MODE can be changed from 0 to one of the following: 1 is for slow clock, 2 is for fast clock, and 3 is for no drift. To remove delay (RTT changes), set the flag INCLUDE_DELAY to 0.

Performance Results

Experienced Loss Rate - Test 1

	10ms	50ms	100ms	180ms
0% Loss	0	0	0	0
1% Loss	1.164047	0.040002	0	0
5% Loss	4.936395	0.380015	0.040002	0
10% Loss	9.996400	1.608129	0.368015	0.052002
20% Loss	19.872795	6.612264	2.808112	0.680027
30% Loss	30.457218	14.405152	7.676614	2.732109

Experienced Loss Rate - Test 2

	10ms	50ms	100ms	180ms
0% Loss	0	0	0	0
1% Loss	1.196048	0.028276	0	0
5% Loss	5.208208	0.440018	0.036001	0.004000
10% Loss	10.260410	1.552124	0.344014	0.040002
20% Loss	20.360814	6.440258	2.700108	0.672027
30% Loss	30.053202	14.284571	7.628610	3.028242

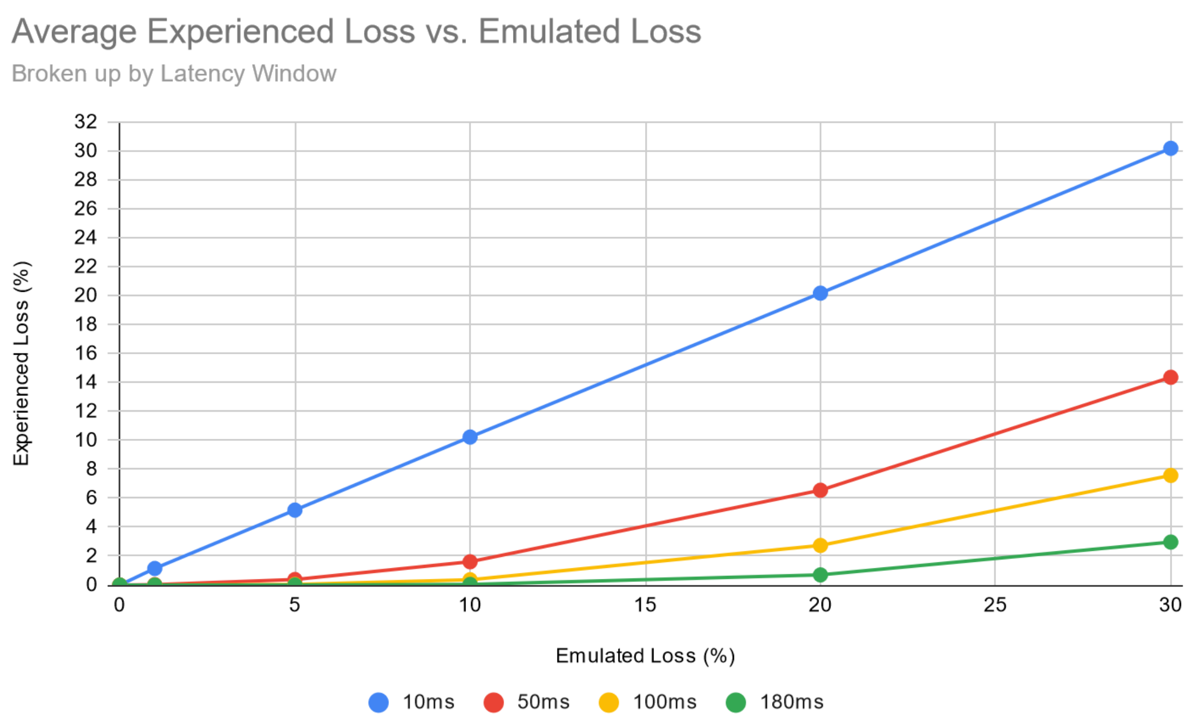
Experienced Loss Rate - Test 3

	10ms	50ms	100ms	180ms
0% Loss	0	0	0	0
1% Loss	1.084043	0.024001	0	0
5% Loss	5.404216	0.328013	0.056004	0
10% Loss	10.440835	1.676067	0.388016	0.040201
20% Loss	20.310078	6.612264	2.708325	0.759101
30% Loss	30.077203	14.404576	7.437328	3.156253

Experienced Loss Rate - Average

	10ms	50ms	100ms	180ms
0% Loss	0	0	0	0
1% Loss	1.148046	0.03075966667	0	0
5% Loss	5.182939667	0.382682	0.04400233333	0.001333333333
10% Loss	10.23254833	1.612106667	0.3666816667	0.04406833333
20% Loss	20.181229	6.554928667	2.738848333	0.7037183333
30% Loss	30.19587433	14.36476633	7.580850667	2.972201333

Graphs



Discussion

Overall, we were satisfied with our implementation. The measured transmission rate at the receiving application was always near the sending transmission rate. Our data did match our expectations. We expected that as our latency window increased, our experienced loss would decrease at an exponential rate. This is what we observed.

We derived a formula for expected loss rate by calculating the probability of receiving a single packet. This is equal to the probability of receiving the initial transmission plus the probability of a successful recovery. The probability of a successful recovery can be further broken down as the sum of the probabilities of a successful recovery on the first, second, third, or any subsequent attempt. We attempt to recover a packet by sending a NACK a total of n times before giving up. For n, we include an epsilon to account for the fact that we have to detect a loss first before we can attempt recovery.

$$n = \frac{latency\ window\ - \epsilon}{1.25 \cdot RTT}$$

$$expected\ loss\ rate\ = loss\ rate * (1 - \sum_{i=1}^{n} \left[(1-(1-loss\ rate)^2)^{i-1}(1-loss\ rate)^2\right])$$

Discussion: Extra Credit Opportunity 1 & 2

Overall, our protocol performed quite well with the streaming of the video from GENI. From 0% loss to 10% loss, a 180ms latency window allowed for a relatively smooth streaming of the yellowstone video. There was occasionally some short, noticeable lag with 10% loss, but it generally performed well with this latency window. For 30% loss, a 180ms latency window did not perform that well; only the audio came through, and it had noticeable stutters. Once the latency window was increased (500ms to 180ms), our protocol performed smoothly again even with a 30% loss. This behavior can be seen in OurProtocolVariousLoss.mp4.

For UDP, the video was relatively smooth with 0% loss. However, even at 5% loss, the video no longer updated and only the audio came through with very noticeable stutters. In order to compare UDP with all the tests we performed in our protocol, we also tested with 10% and 30% loss in the video, and UDP performed even worse in these scenarios. Thus, UDP performed significantly worse than our protocol even on low loss levels. This behavior can be seen in StraightUDPVariousLoss.mp4.

The videos are available in the .zip, and a link to each video is also provided in the README in case there are any issues with the downloaded videos. In each video, we show the protocol running using 0% loss, we then stop both the server and receiver and restart them with 5% loss, we then restart both with 10% loss, and finally we restart both with 30% loss. In the case of our protocol, we restart both again with 30% loss and a greater latency window; with straight UDP, we deliver packets as soon as they arrive, so providing a greater latency window would not change the results.

Discussion: Extra Credit Opportunity 3

Compared to the base protocol, our dynamic protocol performed quite well under all clock combinations for server and receiver (fast/fast, fast/slow, slow/fast, etc.) as well as with RTT changing. Overall, the difference in delivery rate between udp_stream and udp_stream_rcv using our dynamic protocol was more or less the same as what we saw when testing the regular protocol. We also experimented with different possible maximum increments for drift to see if our protocol could handle severe clock drift, and we only really saw decreases in performance - more noticeable differences in delivery rate - when the maximum possible step for drift (MAX_DRIFT_US) exceeded the latency window.