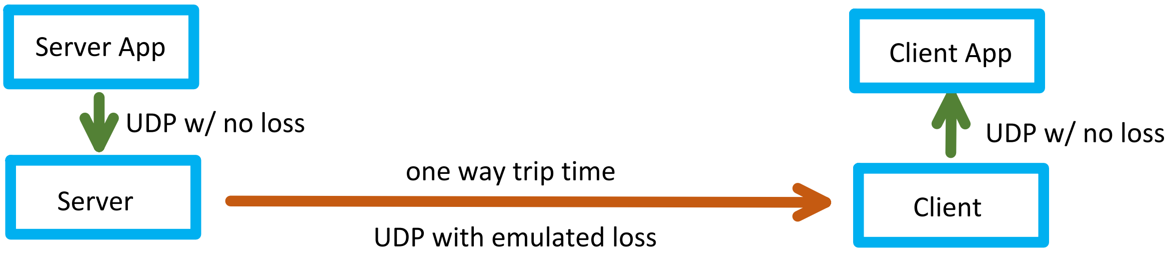
Project 2 Final Design

Monday, October 4, 2021 6:17 PM

Authors: Qingyang Li, Shibo Xing

Main Rationale

In this project we intend to build a real-time data transfer protocol between a server(sender) and a receiver. The transmission is driven by receiver, using NACKs which request individual packets, and CUM_ACKs which acknowledge all the packets up to their sequence numbers. In order to maintain a constant pace in the data transmission, we set a latency allowance that helps the receiver to deliver packets to its application in time. Every packet will contain a timestamp indicating its time of departure from the server. Once a data packet is buffered by the receiver, it checks if the latency allowance has run out since it left the server, more specifically if `now − senderTS > one way delay + clock skew + latency time`, then delivers the timed out packet to the app if the condition is true. Hence, we can ensure that the data stream is delivered in a smooth pace.



Server's protocol

The server uses an ordered mapping of the sequence number of each packet and its timestamp, which designates the departure time from server. It also has an ordered mapping of sequence number and data packet, which is used to cache data from the server application. It has a cum_seq variable to keep track of the latest packets acknowledged so far.

```
unsigned long long int cum_seq;

struct net_pkt {
      time_t senderTS;
      char* payload;
      unsigned long long int seq;
      bool is_end;
};
```

timetable

seq	timestamp
1	136000150
2	136000160
3	136000170

window

seq	*net_pkt
1	0x2414dea0

2	0x2414dea8
3	0x2414deb0

Upon received from app:

Print

When the server program receives an data payload from the app, it goes to the window. The window buffer will keep the existing data packets in memory for future transmission.

Upon received from Receiver (Phase I):

When the server receives an packet containing sequence number 0, it marks timestamp and send it back immediately for the receiver to calculate the delta. It also stores the IP address and socket information at this phase that will be used to block any extra receiver's initiation request. After Phase I, Phase II will begin.

Upon received from Receiver (Phase II):

If NACK:
The server adds a mapping of the sequence number and an ad hoc timestamp, and sends the requested packet. It copies the timestamp information to the data packet if the timetable contains it. It ignores the request if the window doesn't have the sequence number.

If ACK:
The server updates the cum_seq variable. It also erases the both the pairs with sequence number less than the cum_seq from both mappings. Since we use an ordered mapping, we can accomplish this task by repeatedly erasing the smallest sequence pair.

Sender's protocol

The receiver uses similar ordered mappings to as the server, except that the timetable uses the timestamp as the sort key instead of the sequence number. We use this mechanism to accomplish the constant-rate delivery to app.

```
unsigned long long int cum_seq;

struct ack_pkt {
      time_t receiverTS;
      Time_t senderTS;
      unsigned long long int seq;
      bool is_nack;
};
```

timetable

timestamp	seq
136000150	1
136000160	2
136000170	3

window

seq	*net_pkt
1	0x2414dea0
2	0x2414dea8

3	0x2414deb0

Upon received from server (Phase I):

T[illegible]iver calculates and stores the delta as `now - senderT` and updates the cum_seq to 1. Then it begins p[illegible]

Timed task (Phase I):

The receiver sends an NACK packet with seq being 0.

Upon received from server (Phase II):

If the packet's seq is less than cum_seq, ignores it. Otherwise, the receiver caches the packet in the window, and then adds a mapping of its sender timestamp and sequence number.

> *If the packet received is in-order:*
>
> The sender will check the window to find out if more packets sequentially follows the received packet. I t stops when the sequence breaks, and updates the cum_seq to the last in-order packet. Then it sends an ACK with the highest in-order seq back to the server.

Timed task (Phase II):

The receiver will sends a list of NACKs starting from the cum_seq and to cum_seq + window_size, while skipping the ones that are already in the window.

The receiver repeatedly checks if the earliest the sender timestamp cached has expired, using the metric `now - senderTS > delta + latency time` and records the highest sequence number that has expired. While it checks, it also removes the expired from the ordered timetable mapping. It then delivers all packets with a seq less than the highest sequence number that has expired to the app. The delivery will be made in an ordered fashion since the window mapping is sorted by the sequence number. As it delivers, the receive also removes the packets from the window cache.

Evaluation

The graphs below are the data transmission experiment results, using different loss rates and latency windows. We realize that when the latency window is small, the experienced loss rate will be smaller. This is because that due to tight time constraint, many gapped packets are delivered due to time out. The cum_seq is forced to fast forward in this case, causing the data transmission to progress much faster. Therefore, the server doesn't need to process too many retransmission requests in this case, thus having the number of dropped packets reduced.

Video evaluation (bonus 1):

After some experiments, we successfully adapt our receiver timeout and window size with the bandwidth between PITT_NET's domain and geni.uchicago.edu. We find the parameters:

> timeout: 80000ms
>
> W_SIZE: 60

can provide uncongested data transmission.

Loss rate on each svr and rcv (time 2=total loss rate)	0%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets （First Experiment）	0/2500 1	0/2500 1	0/2500 1	0/2500 1
Dropped Packets/Total packets （Second Experiment）	0/2500 1	0/2500 1	0/2500 1	0/2500 1
Dropped Packets/Total packets （Third	0/2500	0/2500	0/2500	0/2500

Experiment)	1	1	1	1
Dropped Packets/Total packets (Average result)	0.0000	0.0000	0.0000	0.0000

Loss rate on each svr and rcv (time 2=total loss rate)	1%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets (First experiment)	1160/26161	1892/26893	1758/26759	1549/26550
Dropped Packets/Total packets (Second experiment)	1174/26175	1696/26697	1645/26646	1766/26767
Dropped Packets/Total packets (Third experiment)	1225/26226	1752/26753	1746/26747	1670/26671
Dropped Packets/Total packets (Average result)	0.0453	0.0664	0.0642	0.0622

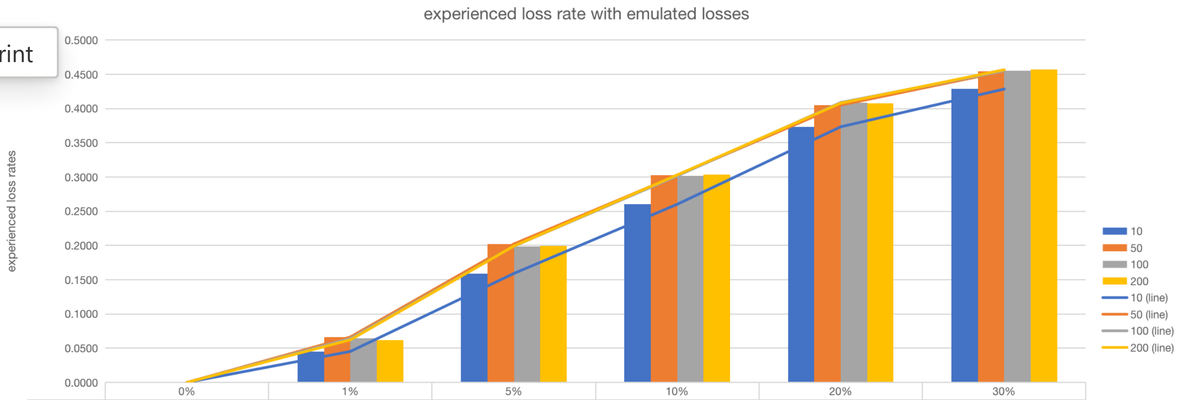
Loss rate on each svr and rcv (time 2=total loss rate)	5%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets (First Experiment)	4801/29802	6445/31445	6020/31021	6057/31058
Dropped Packets/Total packets (Second Experiment)	4692/29693	6143/31144	6458/31459	6230/31231
Dropped Packets/Total packets (Third Experiment)	4699/29700	6431/31432	6154/31156	6364/31365
Dropped Packets/Total packets (Average result)	0.1591	0.2022	0.1989	0.1991

Loss rate on each svr and rcv (time 2=total loss rate)	10%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets (First experiment)	8755/33756	10880/35881	10796/35797	11015/36016
Dropped Packets/Total packets (Second experiment)	8912/33913	10685/35686	10665/35666	10728/35729
Dropped Packets/Total packets (Third experiment)	8726/33727	10986/35987	11006/36007	10892/35893
Dropped Packets/Total packets (Average result)	0.2602	0.3026	0.3016	0.3031

Loss rate on each svr and rcv (time 2=total loss rate)	20%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets (First experiment)	15007/40008	17284/42285	17368/42369	17099/42100
Dropped Packets/Total packets (Second experiment)	14849/39850	16820/41821	17280/42281	17332/42333
Dropped Packets/Total packets (Third experiment)	14837/39838	16953/41954	17225/42226	17236/42237
Dropped Packets/Total packets (Average result)	0.3733	0.4050	0.4088	0.4078

Loss rate on each svr and rcv (time 2=total loss rate)	30%			
Latency Windows (ms)	10	50	100	200
Dropped Packets/Total packets (First experiment)	18890/43891	20978/45979	20846/45847	20996/45997
Dropped Packets/Total packets (Second experiment)	18706/43707	20798/45796	20973/45974	21085/46086

（Second experiment）	[illegible]	[illegible]	[illegible]	[illegible]
Dropped Packets/Total packets （Third experiment）	18633/43634	20705/45706	20799/45800	20947/45948
Dropped Packets/Total packets（Average result）	0.4284	0.4544	0.4550	0.4566



	0%	1%	5%	10%	20%	30%
10	0.0000	0.0453	0.1591	0.2602	0.3733	0.4284
50	0.0000	0.0664	0.2022	0.3026	0.4050	0.4544
100	0.0000	0.0642	0.1989	0.3016	0.4088	0.4550
200	0.0000	0.0622	0.1991	0.3031	0.4078	0.4566
10 (line)	0.0000	0.0453	0.1591	0.2602	0.3733	0.4284
50 (line)	0.0000	0.0664	0.2022	0.3026	0.4050	0.4544
100 (line)	0.0000	0.0642	0.1989	0.3016	0.4088	0.4550
200 (line)	0.0000	0.0622	0.1991	0.3031	0.4078	0.4566

emulated loss rates