Project 2: Simple Window-based Reliable Data Transfer

CS118 - Winter 2016

Taek-Sang Peter Kim
204-271-299

Arvin Nguyen
304-300-135

Introduction

In the laboratory assignment, we designed a reliable data transfer protocol using UDP Sockets in a C++ environment. We implemented a simple congestion window-based protocol built directly on top of a selective repeat protocol, as requested in the specifications. When the client requests a certain file from the server, the server will send the file in a packet-level granularity. Furthermore, we emulated packet losses and corruptions as the actual rate of packet loss or corruption in modern LAN is simply too low.

Implementation

Header Format

The structure of the header of each packet is as such: type, seq, size, checksum. type stands for the type of the packet, ranging from a file request packet, data packet, acknowledgement packet, and a final data packet. seq is the sequence number given in bytes of the packet in the current window. size is the size of the data portion of the packet. checksum is the hash value of the entire packet itself, and it is used to detect corruption of a packet (even though corruption is emulated in our protocol, we decided it was appropriate and better in terms of scalability if emulation was taken away)

Message

Messages about the packets and its header are dumped to the console. Let N be some arbitrary packet number. On the server side, one can see “Sent packet: [Type: 0] [Seq #: 1000] [Payload size: 984]” when a packet is sent. When an ACK is received, the text “Received an ack: Seq # 1000” can be observed.

On the client side, the initial file request is sent and the respective message is “Sent - Type: 0, Seq #: 0, Size: 8, Data: <file_name>”. Afterwards, every time the client receives a packet, it prints out the two messages for confirming receipt and the respective ACK: “Received - Type: 0, Seq #: 1000, Size: 984” and “Sent - Type: 1, Seq #: 0, Size: 0, Data:”.

< Server side console output >

```
cs118@cs118:~/Desktop/shared/CS118-Lab2$ ./server -p 8000  -t 100 -c 0.3 -l 0.3
Starting the server
  - Port:  8000
  - Window size: 5000 bytes
  - Timeout: 100 milliseconds
  - Probabilty of a lost packet: 0.30
  - Probabilty of a corrupted packet: 0.30
Got a request for file: large.txt
File size = 4384913
Required packets: 4385

 > Sent packet:  [Type: 0] [Seq #:     0] [Payload size:  984]
 > Sent packet:  [Type: 0] [Seq #:  1000] [Payload size:  984]
 > Sent packet:  [Type: 0] [Seq #:  2000] [Payload size:  984]
 > Sent packet:  [Type: 0] [Seq #:  3000] [Payload size:  984]
 > Sent packet:  [Type: 0] [Seq #:  4000] [Payload size:  984]
 > Received an ack:  Seq #0
 > Sent packet:  [Type: 0] [Seq #:  5000] [Payload size:  984]
 > Received an ack:  Seq #1000
 > Sent packet:  [Type: 0] [Seq #:  6000] [Payload size:  984]
 - Corrupted ack for packet seq #2000
 > Received an ack:  Seq #4000
 - Lost ack for packet seq #6000
 * Timeout: retransmitted seq #2000
 > Received an ack:  Seq #2000
 > Sent packet:  [Type: 0] [Seq #:  7000] [Payload size:  984]
 > Received an ack:  Seq #7000
 * Timeout: retransmitted seq #3000
 > Received an ack:  Seq #3000
 > Sent packet:  [Type: 0] [Seq #:  8000] [Payload size:  984]
 > Sent packet:  [Type: 0] [Seq #:  9000] [Payload size:  984]
 > Received an ack:  Seq #9000
```

< Client-side console output >

```
cs118@cs118:~/Desktop/shared/CS118-Lab2$ ./client localhost 8000 large.txt 0.1 0.2 5000
 > Sent packet:  [Type: 0] [Seq #:     0] [Payload size:   10]
 > Received a PKT:  Seq #0
 > Sent packet:  [Type: 1] [Seq #:     0] [Payload size:    0]
 > Received a PKT:  Seq #1000
 > Sent packet:  [Type: 1] [Seq #:  1000] [Payload size:    0]
 > Received a PKT:  Seq #2000
 > Sent packet:  [Type: 1] [Seq #:  2000] [Payload size:    0]
 > Received a PKT:  Seq #3000
This packet is corrupted. Discarding...
 > Received a PKT:  Seq #4000
 > Sent packet:  [Type: 1] [Seq #:  4000] [Payload size:    0]
 > Received a PKT:  Seq #5000
This packet is corrupted. Discarding...
 > Received a PKT:  Seq #6000
 > Sent packet:  [Type: 1] [Seq #:  6000] [Payload size:    0]
 > Received a PKT:  Seq #2000
 > Sent packet:  [Type: 1] [Seq #:  2000] [Payload size:    0]
 > Received a PKT:  Seq #7000
```

Timeout

We had difficulty using timers because most timer APIs we found required the use of asynchronous programing method such as multithreading and multiple processes which added a lot of complexity to our program.

Instead we emulated a timer by using a queue. Every time the server sent a packet, the server would record the timestamp and sequence number of the sent packet and push it to the end of the timer.

Because of the first-in-first out property of queues, the front of the queue contains the least recently sent packet. We check if a packet timed out by finding the difference between the current time and the time on the front of the queue. If the difference is greater than a threshold, the packet is considered timed out and resent. The front of the queue is then moved to the end of the queue with the timestamp updated.

Packet Loss & Corruption

Packet loss and corruption are handled independently on both the server and the client. When starting the client and server, one must pass in the probabilities of loss and corruption, that will then be used to determine the rate at which each incoming packet (regardless of its type) will be lost or corrupted. For example, on the client side, if the probability of loss was set as 0.2, a random double r will be generated for each incoming data packet from the server. If the value of r is less than 0.2, then the client will simply regard the packet as "lost".

Window-Based Protocol

The windows maintained by the server and receiver for the selective-repeat protocol were also implemented using queues.

In the server, each element in the queue keeps track of the sequence number of the packet sent and whether or not that packet has received an ACK. The window is "slided" by dequeuing as many elements as possible from the front of the queue that are marked as having received an ACK. Packets are then sent and added to the queue until the queue's length reaches the window size limit.

In the client, each element of the queue keeps track of the sequence number of the packets it expects to receive. Once the client has received a packet, it saves the payload in the corresponding element in the queue. the client then dequeues as many elements as possible that have payloads and writes to the file.

Congestion Control

We implemented slow start and congestion avoidance. We did not time to implement fast recovery because that it would require reimplementing a whole new protocol.

Challenges Faced

We found the documentation for creating UDP sockets to be not easy to understand. It was also difficult to implement selective repeat with multiple timers (one timer per packet sent), so we simplified the problem by using only one timer for all the packets. Testing the programs thoroughly was a challenge because of the asynchronous nature and nondeterministic nature of the programs when lost or corruption is introduced. To debug we had detailed and colored printouts for each event.

Conclusion

Overall, both me and my lab partner have gained some valuable hands-on experience by implementing a relatively complex protocol of selective repeat.