Project 2: Reliable UDP

Author: Anup Bharadwaj Date: 10/11/15

Introduction:

Reliable UDP is an application built on UDP which incorporates the features of TCP without the overhead of the connection establishment phase. The application sends duplicate acknowledgements indicating a lost or out of order packet, estimates RTT using Jacobson/Karels algorithm to retransmit the packet upon timeout and has congestion control supporting slow start and congestion avoid phases.

Header design:

The header of the Reliable UDP consists of the following fields:

- Len (int): Denotes the length of the data in the payload in bytes
- Ack_indicator (int): Indicates whether the packet is a data, acknowledgement or a close indicator. The possible values in this field are: 0 – indicating that the packet is an acknowledgement, 1 – indicating that the packet contains data and -1 – indicating that the connection needs to be closed.
- Seq_num (int): Contains the sequence number of the packet. It denotes the byte number of the first byte of data.
- Ack_num (int): Contains the acknowledgement number. The client acknowledges each packet by sending the number of the next byte to be sent to the server. There could be cumulative acknowledgements of multiple packets when there is an out of order packet being buffered.
- Data: Contains the data to be sent. The maximum data size in this application is set to be 100 bytes. Thus, the application can send at most of 100 bytes of data at once. Having 100 bytes limit helps us to understand the flow of the program better as the features of the program could be better analyzed. With a small packet size, even a small file transfer involves a considerable number of packets.

The maximum total length of each packet including the header would thus become:

Data (100bytes) + 4 * sizeof (int) bytes

Implementation:

The application consists of a server and a client module. The advertised window is negotiated offline and passed as a command line parameter. Server takes port number, drop probability (0 to 1) and window size as command line parameters and client takes the hostname, port number, filename (to retrieve from the server), probability of introducing high latency communication and window size.

Based on the parameters passed, the client sends a request for the file by including the file name in the data field of the header to the server and the server begins transmitting the files in packets. The number of packets sent at once is governed by the congestion and receiver window sizes. Once the server finishes sending the packet, it sets the finish flag indicating the end of file transfer and returns to listening to new file requests. The client, upon reading the flag, closes the connection. In case the file is not found, the server simply closes the connection by sending the close signal.

The server probabilistically drops some packets and the client once recognizing that the packets could be lost or out of order, buffers them and sends duplicate acks. Once the right packet is received, the client sends a cumulative acknowledgement of all the packets received so far. The server on the other hand, calculates RTT using sample RTT, updates the congestion window and waits for the acknowledgement. Once the packet times out, the server re transmits the packet and resets the congestion window to 1 MSS.

Packet header:

Header is implemented as a structure in C; the definition of the header is given below:

```
struct rel_udp_header
{
int len;
int ack_indicator; //0=acknowledgement 1=data -1:close
int seq_num;
int ack_num;
char *data;
};
```

The packets are sent as a character stream with “:” as delimiter among the server and the client. Once they are received, they are parsed based on this delimiter and the above structure is formed.

Duplicate Acknowledgements:

The server waits for 3 duplicate acknowledgements before it decides to resend the missing packet. It is implemented using a simple counter. The flow is handed back to the function once the counter turns 3 and the index of the file is set to the “(ack_num – 1)” of the acknowledgement packet.

Client side buffering:

To buffer an out of order packet, a linked list is implemented which adds the incoming packets to be buffered and sorts them based on their sequence number. The application pops these packets sequentially and displays the data once the packets are in order. A cumulative ack is sent to the server.

Timeout and RTT:

To ensure that the server listens for acks only for a specified period of time, the socket is passed with options for the “recvfrom” function to ensure that the server listens for acks only for some period of time. The RTT is calculated using Jacobson/Karel’s algorithm. If “recvfrom” returns a negative value after listening for acks, it means timeout has occurred. Now the program does a re-transmission of the packet.

Congestion control:

The window size is determined to be the minimum of cwnd and rwnd values. While rwnd is negotiated offline, the cwnd variable starts with the initial value of 1MSS. Adopting the congestion control mechanisms employed by TCP, the application begins with the “slow start” phase where it increases the congestion window by 1 MSS for every acknowledgement received. The program enters “congestion avoidance” phase once the cwnd becomes greater than ssthresh value. A timeout of a packet immediately results in cwnd being reset to 1MSS and shifts back to “slow start phase”. The program outputs the percentage and number of packets transmitted in each phase.

Latency and dropping packets:

The server and the client has a function which returns a Boolean value based on the probability value supplied. Using this, a packet drop can be simulated at the server side where the server will not send the packet based on the probability of the drop given. The client can simulate high latency communication by taking a probability value and sleeps for 100 milliseconds before processing the packet.

Conclusion

With ReliableUDP, it is possible to send files more reliably with flow and congestion control without the overhead of connection establishment. By simulating different levels of congestion in the network and high latency communication at the client side, ReliableUDP was still able to transfer the complete file in order by packet retransmission, duplicate acknowledgements, buffering out of order packets etc. Timeout instances were more when there was high latency communication at the client thus slowing down the packet transmission rate. More packet drops resulted in out of order packets received by the client and resulted in more instances of duplicate acknowledgements. Both packet drops and high latency at client affected the sending rate. Increasing the receiver window gave room for more packets to be sent at once from the server.