Design and Implementation of a Reliable Transport Protocol

CS 3251

20 April 2016

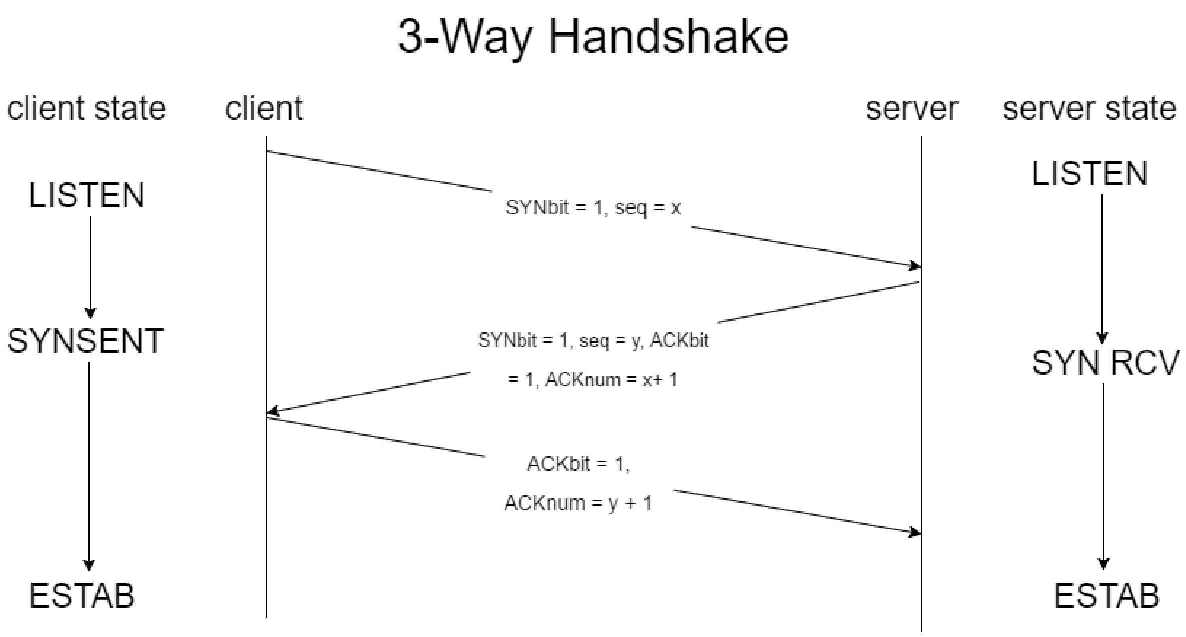
Enmao Diao, emdiao@gatech.edu, ediao3

Jordan Zheng Xi Kwong, jkwong3@gatech.edu, jkwong3

High Level Description

A Reliable Transport Protocol (RTP) is a reliable, in-order stream of data that fixes packet losses and duplicate packets, and reorders packets. The protocol implemented is the sliding window, which can significantly improve performance.

For the sending of data to the receiver, the sender can drop, reorder, delay and possibly duplicate packets. The receiver wants to receive the packets in the order which the sender sends them, and only wants exactly one of each of the packets. The protocol is connection oriented, which requires a connection to be established between the receiver and sender before any data packets can be transmitted. As such, a RTP socket is used to establish the connection, done by a three-way handshake:



At first, the client's and server's states are both closed. To establish a connection, the client must first send a SYN to the server, and receive an ACK from the server. The server sends a SYN + ACK, and the client is now done with the connection establishment. After transferring all of the packets, the connection will be closed simply by disconnecting.

The UDP packets encapsulates the RTP packets. The packets are then sent, packet by packet to the receiver until the all of the packets are received by the receiver. A window size argument is used by the receiver to inform the sender so that the sender will not overwhelm the receiver. This means that the sender will not be allowed to send more than the window size and has to wait for the acknowledgement of packets before the sender can shift the sliding window by the correct number.

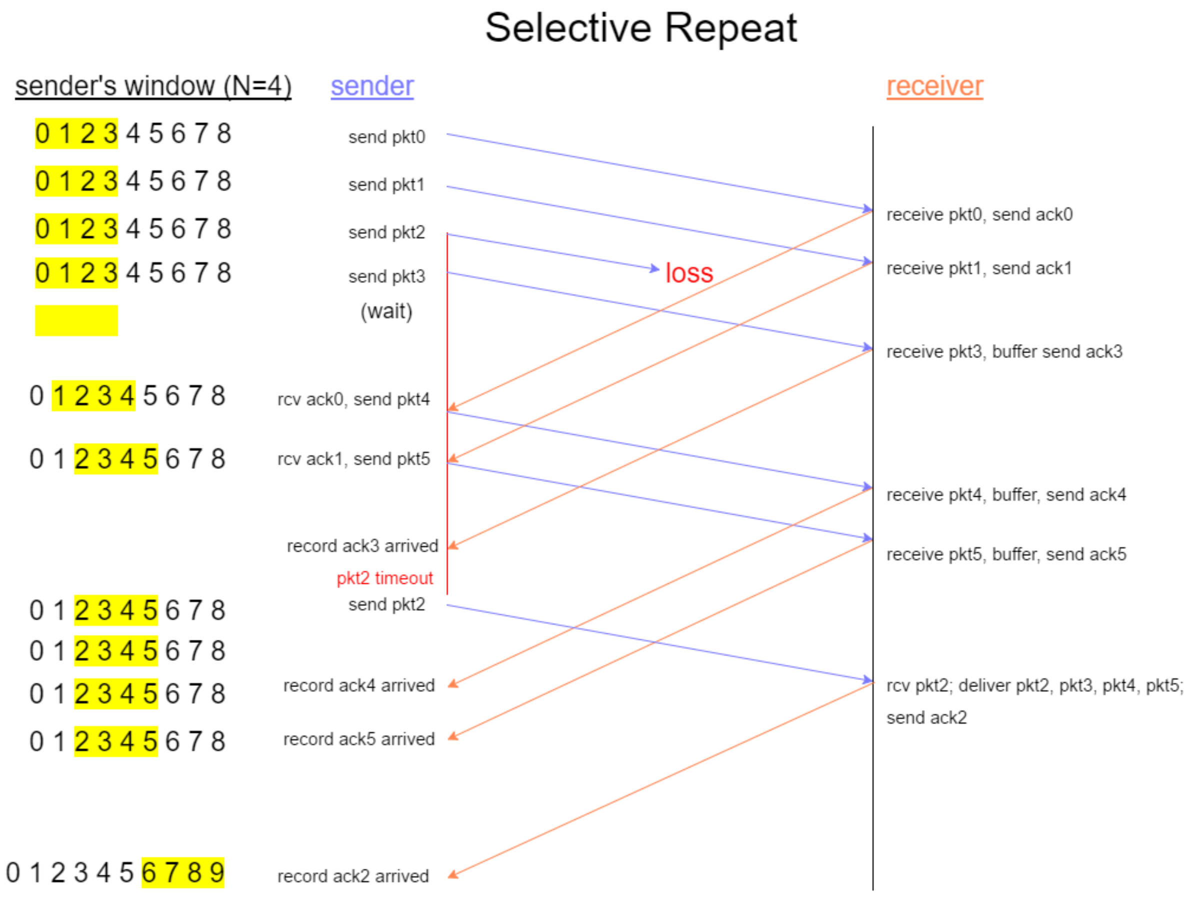
This is how the sending of the packets is done:

1. Sender sends the packet to the receiver, along with the sequence number attached in the packet.

2. Receiver receives the packet, and checks the checksum of the packet to see if the packet is corrupted. If the packet is corrupted, the packet will be dropped and after the duration of the

timeout, the packet will be sent to the receiver again. If not, the receiver sends an acknowledgement back to the sender, along with the sequence number of the packet.

3. The pipelining of the sending of packets is done by Selective Repeat. It allows the sender to have up to N unacknowledged packets in the pipeline, where N is the sender's window. The receiver individually acknowledges all correctly received packets, and sender only resends packets if the ACK was not received. The sender maintains a timer for each unacknowledged packet so that when the timer expires, it will retransmit that unacknowledged packet. This is shown here:



For Selective Repeat, packets need to be sent consecutively. In the figure above, pkt2 experienced a timeout. The sender's window cannot be shifted until the packet is sent to and acknowledged by the receiver. Only when pkt2's acknowledgement is received by the sender, then the sender's window can be shifted. When ack2 finally arrives, the sender's window shifts from “2 3 4 5” to “6 7 8 9”. This allows the sender to start sending the packets 6, 7, 8 and 9.

4. This is repeated until all of the packets are sent. The last packet will indicate a FIN bit that tells the receiver that that is the last packet and there will not be any more transmission of packets.

Congestion Control

Congestion control is implemented to prevent lost packets and long delays. It is implemented using the following formula:

Transmission window size = min(advertised window size, congestion window size)
It uses slow-start from 1MSS and when it reaches threshold cwgn = cwgn + 1/cwgn

Duplicated Packets

When there is a packet that shows a sequence number that is smaller than the first packet of the receiver's window, it is a duplicated packet. The duplicated packet will be dropped.

Corrupted Packets

All of the packets are checked using a java class Cyclic Redundancy Checksum (CRC32), where the packet's checksum is compared before and after transmission. If the CRC32 values are not the same, then the packet is corrupted. The corrupted packet will be dropped, causing a timeout. The sender then retransmits the packet that caused the timeout.

Lost Packets

When there is no ACK received after the duration of the timeout, it indicates that the packet is lost. The sender then retransmits the packets that has not been acknowledged, which is the first packet in the sender's window. The sender is only allowed to send the number of packets that is equal to the receiver's window's size, where the size of that window is indicated together with the acknowledgement which the receiver sent before.

Re-ordered Packets

Packet that are out of order are detected by the receiver. For an out of order packet, it is not the first packet of the receiver's window. The receiver will wait for the sender to send the packet that is the first packet of the receiver's window before sliding the receiver's window. For packets that are out of the receiver's window, they will be dropped.

De-multiplex data to different RTP connections at the same host

This is done by creating threads that use a Java class HashMap<K,V>, where the key is the socket address which is a combination of IP address and port number, and the values are an array of objects that stores the values of startWindow, windows_ack, buffer_rcv, ifFIN, numberOfTimeouts, windowSize, windows, Windowslist, queue, lock, maxSenderWindowSize and sendSeq.

The server program is multi-threaded. The multi-threaded server creates a thread for every communication that it accepts from a client. The server can still continue to listen to requests from other clients and starts communicating with other clients by creating a new thread.

Bidirectional Data Transfer

When the connection is established between the sender and the receiver, data can be transferred between both sender and receiver in both directions. The use of threading support this feature automatically, as the sending and receiving packets are done separately and independently.

Byte-Stream Semantics

The file is made up of data with a large number of bytes. As such, all of the RTP packets can be transformed into bytes, and each RTP packet consists at most 1000 bytes. The last packet indicates a FIN flag to signal that it is the last packet. The packets of data will be re-joined to form the file at the receiver's end.

Special Parameters in Design

A timeout class PacketTimeout is used so that it starts the timer when each packet has been sent. This calculates when a packet has been timed out.

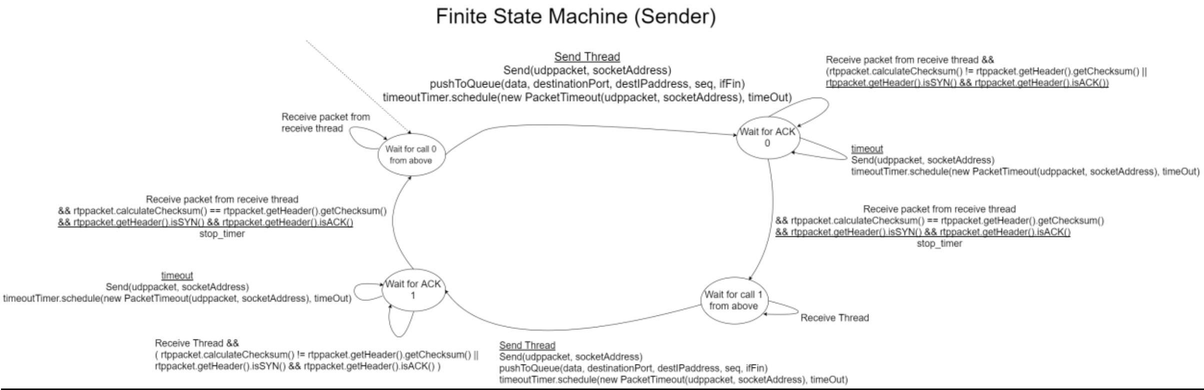
RTP Header

<table>
<tr><td>31</td><td colspan="2">RTP Header</td><td>0</td></tr>
<tr><td colspan="4">Source Port (32 bits)</td></tr>
<tr><td colspan="4">Destination Port (32 bits)</td></tr>
<tr><td colspan="4">Sequence Number (32 bits)</td></tr>
<tr><td colspan="4">Data Length (32 bits)</td></tr>
<tr><td colspan="4">Checksum (32 bits)</td></tr>
<tr><td>ACK</td><td>SYN</td><td>FIN</td><td></td></tr>
<tr><td colspan="4">Receiver’s Window (32 bits)</td></tr>
</table>

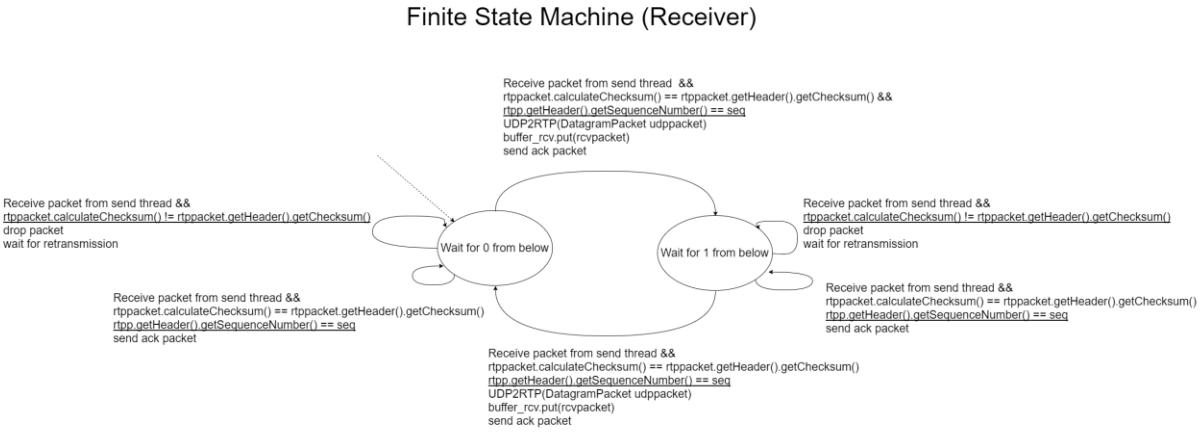
The RTP Header Fields is structured as such:

	Bits
Source Port	0 – 31
Destination Port	32 - 63
Sequence Number	64 - 95
Data Length	96 - 127
Receiver’s Window	128 - 159
Checksum	160 - 191
ACK, NAK, SYN, FIN flags	192 - 194
Receiver’s Window	224 - 255

Finite State Machine (Sender)



Finite State Machine (Receiver)



Formal Description of the Protocol's Programming Interface

We use multithreading for both sending and receiving. For each host, there is a thread to receive incoming packets. For each connection, there is a thread to send packets stored in the queue. In the application level, the user simply call startReceive() to start the receiving thread. When it needs to send, it will first use connectionSetup() to do three-way handshaking and then the sending thread will also be setup on both ends so that the two hosts can send packets over their own thread. This contributes to demultiplexing because each client will have its own sending thread with the server. The server will hear from all incoming packets and deal with them based on their flags and data. Moreover, having two threads for sending and receiving respectively also supports bidirectional transmission.

startReceive()	start the receiving thread
ConnectionSetup()	three-way handshaking
getoutput()	get an array of packets separated with FIN flag