CS3251 Computer Networking I

PRJ 2 - Reliable Transfer Protocol

November 25, 2015

Elliott Childre	rchildre3@gatech.edu	rchildre3
Derrick Williams	derrickw@gatech.edu	dwilliams306

Introduction

Our Reliable Transfer Protocol (RTP) will sit atop UDP to provide a reliable way to connect and transfer data between client and server. The protocol will be pipelined with multiple packets in flight, will be reliable by minimizing or eliminating errors with the packets, provide flow control, and maintain state information for each connection. RTP will essentially be replacing TCP, but with the difficulties of dealing with UDP.

The state machine, header, and window usage will mimic TCP as closely as possible, while removing some of the parts that are not needed for us (e.g. header length in header due to a fixed header size and other non-essential flags).

Reliability

Our RTP transport protocol will meet the requirement of being reliable by providing each of the following services:

Lost Packets: Acknowledgment packets will be used to notify the sender of correct transmission. If a packet is lost, the sender will not receive such acknowledgment. This will be handled by a timeout function to resend the packet.

Corrupted Packets: Corrupted packets will be checked using the header provided by RTP, which will trigger the receiver to send a NACK and drop the packet.

Duplicate Packets: Any extra packets that are received will be dropped, packet will be uniquely identified by their sequence number.

Ordering of Packets: The ordering of packets will be maintained through the use of sequence and acknowledgment numbers that are relative to the packets themselves.

Flow Control: Flow control will be maintained by the use of a receiving window field in the RTP header relative to the number of the packet, which enumerates the number of bytes the sender can receive.

Checksum: As mentioned in the corrupted packets section, the RTP will be using the checksum in the RTP header to determine any bit errors or other corruption. The Checksum will be calculated on the th entire packet, including the header, so the flow of the sender will be to 1) construct the packet + header, 2) zero out the header field, 3) calculate the checksum, and 4) place the checksum in the header and send the packet. The flow for the recipient will be to 1) read in the checksum from the header, 2) zero out the checksum field in the header, 3) calculate the checksum, and 4) drop or accept the packet depending on the results of the checksum. Corrupted packets will be dropped and trigger a NACK response.

Packet Size: The packet size will be limited to approximately 1,456 bytes which is the typical maximum size of the payload of UDP (1,472 bytes) minus the RTP header (16 bytes). The corruption rate of the packets will be the same across the data stream, but will less errors will occur per packet if we keep the data to a more manageable size such as the normal UDP payload size instead of the theoretical size of ~65,536 bytes minus header information.

RTP Header

Sequence number (32 bits): A random number will be generated for the starting packet and subsequent packets will use that number plus the number of bytes that were in that previous packet.

Acknowledgement number (32 bits): The Acknowledgment number will be used by the receiver to notify the sender of the next packet that the receiver is expecting. For example if packet X was correctly received by the receiver (without checksum error), and processed, then ACK(X+1) would be sent.

Checksum (16 bits): The sum of the byes in both the payload and header without the checksum field, modulo with the maximum number in the checksum field (2^{16} - 1)

Receiving Window (32 bits): The receiving window size will be used to convey to the receiving host that the sending host can receive up to the receiving window size data limit in this header before data may start to be dropped at the sending host's buffer.

Flags (1 bit each, 8 bits total, lower 4 bits are used in implementation)

ACK: acknowledgement flag to tell the receiver that the packet was received and everything is well with the packet information that was sent.

SYN:synchronization of client and server to initiate communication.

FIN: signals a final packet of a transmission, which triggers a connection teardown.

NACK: signals a non-acknowledgment, the recipient is instructed to resend the specified sequence.

IP Address (32 bits): sender's IP address to be located in packet so that the receiving host can determine where the packet came from; host will never see the UDP header.

Port Number (16 bits): sender's Port number to be located in packet so that the receiving host can determine where the packet came from; host will never see the UDP header.

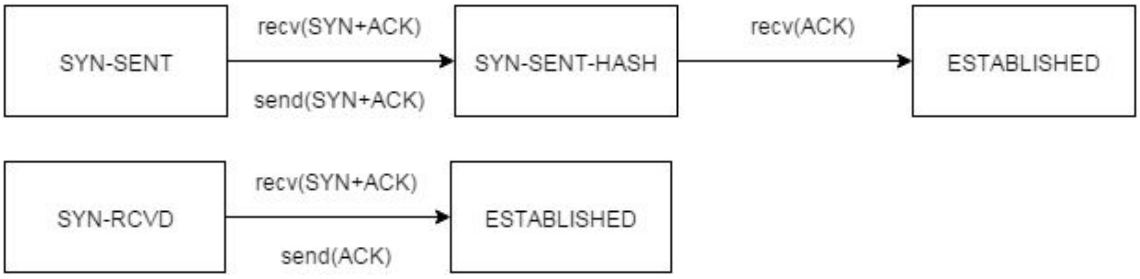
Optional Header (24 bits): They will be 12 bits of miscellaneous flags or other data for any additional options needed for final implementation. May delete; as of today(11/13) the header doesn't include.

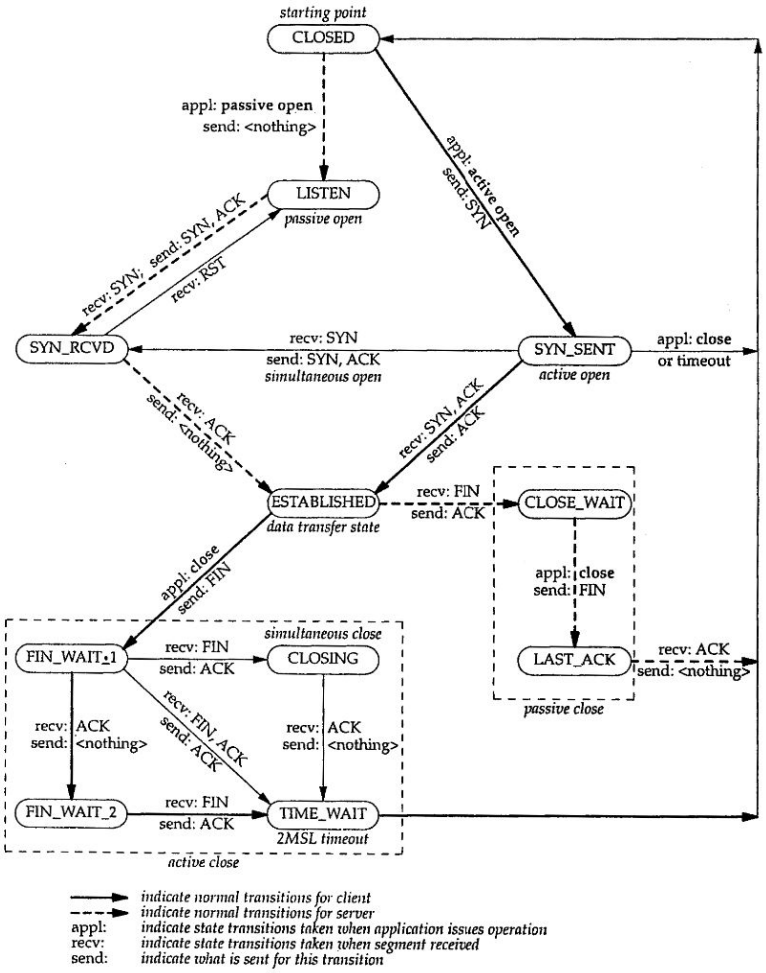
Total Size (24 bytes): Total size of RTP header will be 4 bytes for the Sequence Number, 4 bytes for the Acknowledgement Number, 2 bytes for the Checksum, 4 bytes for the Receiving Window, 4 bits for flags, and 12 bits of Optional Header information for a total RTP header size of 16 bytes.

Finite State-Machine

The RTP will use a similar state machine as TCP to handle the connection and closing of connections between clients and server. During the established period when either client or server can send information, additional state information will be maintained in determining if the client is getting information from the server or posting information to the server and at what stage of the process is at. Each client's state will be maintained by using an array of connection objects and constantly monitoring incoming and outgoing packets to alter state. The following states will be used between the client and server, including one extra state called SYN-SENT-HASH that resides in between the SYN-SENT and ESTABLISHED states, as well as the following edit to the SYN-RCVD to ESTABLISHED transition:

Add CLOSED state for bad challenge hash





Bi-Directional Data Transfer

RTP will be able to handle 2-way transfers by implementing the sending and receiving functionality on both the client and server side applications and allowing the client to download and upload data to the server.

API for Application Layer

The API for RTP will include all of the following functions:

<table>
<tr><th>Client</th><th>Server</th></tr>
<tr><td colspan="2">socket()
Returns a socket across which messages can be sent or received</td></tr>
<tr><td rowspan="2">connect()
Establishes a connection between the given socket and the server socket</td><td>bind()
Assigns the local Internet address and port number for a socket</td></tr>
<tr><td>listen()
Indicates the given socket is ready to accept</td></tr>
</table>

<table>
<tr><td rowspan="2"></td><td>incoming connections</td></tr>
<tr><td>accept()
Blocks and waits for connections from client addressed to the IP address and port number to which the given socket is bound</td></tr>
<tr><td colspan="2">send()
Sends the bytes contained in the buffer over the given socket</td></tr>
<tr><td colspan="2">recv()
Receive the bytestream from the specified address</td></tr>
<tr><td>close()
Close the connection</td><td>close()
Terminates communication with current client and continues onto next client</td></tr>
</table>

Functions Within RTP

addConnection(): Adds a new connection object to the set of open connections.

endConnection(): Removes the connection from the set of open connections.

timeout(): Initiate timeout for each packet sent (Initialize time to 5 seconds). Initiate timeout for window of packets (Initialize time to window size * a set time frame per packet).

dropPacket(): If a packet is a duplicate or corrupted, packet will be dropped.

calculateHash(): Calculate MD5 hash of challenge question answer.

checkHash(): Check MD5 hash of challenge question with randomly generated string of characters.

calculateChecksum(): Sum of the bytes in the payload and header (without the checksum) modulo with the maximum number possible in the checksum field (2^{16} - 1)

checkCecksum(): See calculateChecksum()

send(): Use the functions listed above to send out a packet and start the timeout timer

receive(): Use the functions listed above to receive a packet, send the correct acknowledgment, and order the data into a bytestream for the application layer.

slidingWindowSize(): Determine appropriate window size for transmission. Will be evaluated continuously throughout data transmission.

Sliding Window

RTP protocol will implement a sliding window protocol similar to TCP for managing sent packets and flow control. A window size of 1 will act as a Stop-And-Wait ARQ protocol, while anything higher will be considered pipelining the data packets. The window size will convey how much data can be received by the sending host before packets start to be dropped. The window size will be limited initially to 100 packets, then increases by 5 packets per each successful window acknowledgment. If window is timed out the window size will be halved to prevent network congestion. A more sophisticated algorithm may be used during implementation.

Selective Repeat

If a particular packet is received as corrupted, the recipient will send a NACK back to the sender in hopes that the sender will be able to receive the NACK and resend the packet before the window times out causing the costly halving of the window size.

4-Way Handshake Communication Setup

To prevent denial-of-service (DOS) attacks on our protocol and to improve on the TCP protocol concept, we will be implementing a 4-way handshake for connection setup between the client and server instead of the standard 3-way TCP handshake. The client will initially send a synchronization request to the server. The server will send back a challenge question to the client. The client will then send an answer to the challenge question as a MD5 hash. The server will receive the MD5 hash and check the answer. If everything is correct, the server will send back an ACK verifying the connection is made. If anything is incorrect, a NACK will be sent back to the client. See figure below for illustration of the 4-way handshake for setting up connection between client and server.