PA 2 - Design and Implement a Reliable Transport Protocol
April 20, 2016

James Leopold jleopold3@gatech.edu
Kate Unsworth kunsworth@gatech.edu

FILES SUBMITTED

dbclientRTP.py	relational database client
dbengineRTP.py	relational database server
ftaclient.py	file transfer application client
ftaserver.py	file transfer application server
README.pdf	readme
rtp.py	reliable transfer protocol source code
Sample.txt	sample output file
3251.jpg	image for file transfer application

INSTRUCTIONS FOR COMPILING AND RUNNING PROGRAMS

The applications may be run from the command line as follows:
    FTA Server: python ftaserver.py <port> <rwnd>
    FTA Client: python ftaserver.py <host>:<port> <rwnd>
    RDBA Server: python dbengineRTP.py <port>
    RDBA Client: python dbclientRTP.py <host>:<port> <gtid> <attrs>

While running FTA use the following command line arguments:
    get <filename> downloads a file from server
    get-post <filename1> <filename2> downloads file 1 and uploads file 2 to server
    disconnect closes connection with the server

Notes:
    All code is written in Python 2.7.
    Attributes to request for RDBA client should consist of one or more of the following separated by spaces:
        first_name
        last_name
        quality_points
        gpa_hours
        gpa

DESIGN DOCUMENTATION

I. How RTP works
    A. Connection establishment
       RTP establishes a connection with a three-way handshake (similar to TCP). First the server must bind to a port and listen for a SYN from the client (calls the RTPSocket bind method). The client requests a connection (calls the RTPSocket connect method) by sending a SYN to the server with

seqnum=client_isn (a randomly generated integer). The server (calling the RTPSocket accept method) receives the SYN and sends a SYNACK to the client with seqnum=server_isn (randomly generated integer) and acknum=client_isn+1 to grant the connection. Last, the client replies with an ACK to acknowledge the connection. The ACK has seqnum=client_isn+1 and acknum=server_isn+1. When the three-way handshake is completed, the server and client each return an RTPConnection object.

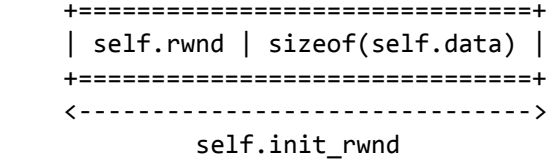
B. Connection termination

RTP terminates a connection with a four-way handshake (similar to TCP but with separate clientClose and serverClose methods). The client calls clientClose, which initiates the connection termination by sending a FIN to the server. The client then starts a timer and waits to receive a FINACK from the server indicating that it has received the FIN and will also begin connection termination. Meanwhile, the server has called serverClose. After the server receives the client's FIN, it sends an ACK to the client, followed by a FIN. When the client receives the ACK from the server, the client then waits for a FIN, and sends an ACK back to the server when the FIN is received. The client waits 2 * the RTT of an RTP packet to make sure the ACK to the server gets received (when this happens, the server closes its side of the connection). Then the client closes its side of the connection.

Note: A connection from the client side is terminated by closing the UDP socket (since a client may connect to only one server), while the serverClose method closes a connection by setting the RTPConnection object to off and deleting the RTPConnection from self.connections (the hash map holding all the connections with clients).

C. Flow control

Each RTPConnection has a receive window (self.rwnd) which is the number of bytes that may be sent to that connection. If there is no data in the receiver buffer (self.data, the data that has been received in a connection but not yet sent to the application layer), this will be equal to self.init_rwnd, which is the maximum number of bytes that may be sent to that connection at a time. If there is data, self.rwnd will be equal to self.init_rwnd - sizeof(self.data), that is, the remaining available space in the receiver buffer.



The Go-Back-N protocol is used for flow control. Go-Back-N is implemented here as shown in the textbook. For RTP, a congestion window self.cwnd is used on the sender side (see the section on Congestion Control) so self.cwnd is also used to determine N, the number of packets that may be sent at a time.

SENDER SIDE (RTPSocket send method):

The sender initially sets two integer variables self.base and self.nextseqnum to 0. Self.N is initialized as the floor of the minimum of (rwnd of

connection, cwnd of connection/MSS). We set self.N to the minimum of the two window sizes to prevent either the sender or receiver buffer from overflowing. If nextseqnum is less than self.base + self.N, this means the nextseqnum is within the send window, so the packet is sent. Otherwise it is not sent. After a packet is sent, the sender waits for response ACKs from the receiver. When the sender receives a cumulative ACK (indicating all packets up to that acknum have been received), it increments base by the acknum + 1. This moves the send window up to begin sending the next N unACKed packets. If the timer for the first packet in the current send window times out before an ACK is received, the send window remains the same and the packets in that window are re-sent.

RECEIVER SIDE (RTPSocket recv method):
On the receiver side, when the receiver receives a packet from the sender, it checks if the seqnum of the packet is equal to the seqnum it expects to receive next (expectedseqnum). If so, the packet is appended to the receive buffer (self.data for the receiving connection), expectedseqnum is incremented by 1 so the next packet can be received, and an ACK is sent for that packet. The size of self.rwnd is recalculated based on the size of the data placed in the buffer. If the seqnum is not equal to the expectedseqnum the packet is ignored. This prevents packets from being received out of order.

If a packet has eom=1, the receiver handles it differently because this indicates that the packet is at the end of a message. Since RTP is connection-oriented, the eom value is used to tell the receiver where one message ends and another begins.

D. Duplicate packets
Packets have sequence numbers and acknowledgement numbers. The RTP receiver extracts the data from a packet only if the sequence number of the packet is equal to the expected sequence number (the sequence number of the next packet that should be received). Otherwise, the ACK for the most recently received in-order packet will be re-sent, causing the sender to re-send the packets in the send window. The expected sequence number is incremented by 1 when the expected packet is received and the ACK is sent. If the sender sends a duplicate packet (one that has already been received), the seqnum of that packet will not equal the expected seqnum so the receiver will discard it.

E. De-multiplexing
RTP uses connection de-multiplexing to allow multiple clients to communicate with the server simultaneously. Connections are represented by RTPConnection objects. When a server side application is started using RTP, there is a single UDP socket which can be used to send and receive messages to different clients. Every RTPSocket has the following variables which are used for de-multiplexing messages to different clients:

self.connections: a hash map containing key-value pairs where the key is the connection id in the form of (host, port) and the value is the RTPConnection object for that address.
self.ackList: a hash map containing key-value pairs where the key is the connection id and the value is a list of ACK packets received for that connection

self.finList: same as self.ackList but for FIN packets
self.SYNqueue: queue of SYN packets received. The connection from which the SYN was received can be determined by the packet header.
self.SYNACKqueue: same as self.SYNqueue but for SYNACK packets

There is also a lock for each of these variables. The lock must be used whenever accessing the variables are accessed, as it prevents them from being modified at the same time by different client threads.

When an RTPSocket is initialized, the RTP recv function is called in a new thread. This recvThread continues running until the program exits (as it is a daemon thread). Inside the recv function, the RTPSocket continuously receives packets and places them in the appropriate hash maps and queues depending on the connection from which they were received. In the send function, packets are sent to their intended destinations (addr, passed in as an argument) by indexing into self.connections at addr and sending through the connection corresponding to that address.

A client application will also have a recvThread running with a connection to the server, but since clients don't need to connect to multiple servers there is no demultiplexing to multiple clients (though the client still has a self.connections, etc.)

F. Byte-stream semantics

RTP assigns sequence numbers to packets to make sure they are delivered in the correct order. When the RTP receiver receives a packet it sends an ACK to the sender with the acknum (acknowledgement number) indicating which packet the ACK is for. The receiver discards any packets where the sequence number is not equal to the expected sequence number. This prevents the receiver from receiving duplicate packets or out-of-order packets. RTP connections have send and receive buffers to hold packets that are waiting to be sent to the receiver or are ready to be sent from the receiver to the application layer.

G. Special values/parameters

1. isACKed:

RTP packets have an isACKed value to indicate whether the sender has received an ACK indicating that the packet has been received.

2. eom:

This stands for "end of message" and is a RTP header field. If eom=1, the packet is the last packet in a message so the receiver knows to return the message data to the application layer.

Other special values/parameters are described throughout this design documentation.

H. Packet Losses

Lost packets are handled in Go-Back-N. If a packet is lost, it will not be ACKed by the receiver. This results in a timeout at the sender, causing the sender to re-send all packets in the sliding window starting after the most recently ACKed packet. So the lost packet will be re-sent.

I. Re-ordered Packets

Re-ordered packets are handled with sequence numbers and acknowledgement numbers as described in the section about byte-stream semantics. If a packet is sent out of order, the receiver will discard it rather than ACK it because the sequence number is not the number expected. The sender will time out and re-send the packets in the sliding window.

J. Corrupted Packets

A checksum is used to check for corrupted packets. The checksum for a packet is computed on a pseudo-header plus the packet data. The pseudo header consists of all the RTPHeader fields except len_data and checksum. It is computed by first concatenating the packet data and the packet header (as a byte string). The checksum is initialized to zero and padded at the end if its length is odd. Then the characters are looped through and added to the checksum which is then folded and inverted. The bit operations done in the checksum algorithm are the same as in TCP (source: http://www.binarytides.com/raw-socket-programming-in-python-linux/).

The RTPPacket class has a getChecksum method and a setChecksum method. getChecksum calculates and returns the checksum value of the RTPPacket. setChecksum actually calls getChecksum on the packet and sets its checksum header field equal to the value returned by getChecksum().

When a packet is created in the RTPSocket send function, its checksum field is set with setChecksum. In the recv function, the checksum is used to check for corrupted packets. When a packet is received, the checksum value from its header field is compared to the value calculated with getChecksum for the packet. If the calculated value does not match the header checksum, the packet is determined to be corrupt and it is discarded by the receiver. It will be re-sent when the sender times out.

K. Congestion Control

Congestion control in RTP is similar to TCP congestion control. It is done at the connection level - each connection has its own congestion window. Loss events are indicated by timeouts (when the sender sends a packet but does not receive an ACK before the timer runs out). The sender will never send more data than the minimum of (rwnd, cwnd) for a connection to avoid overflowing the sender or receiver buffer. The receiver has a congestion window of self.cwnd bytes. This is the amount of data that is allowed to be sent at one time. Initially when a connection is created cwnd = 1 MSS. Another variable, self.ssthresh, is set to 25000 bytes (an arbitrarily large number). Cwnd is changed every time an ACK is received for a sent packet. While cwnd < ssthresh, cwnd is multiplied by 2 every time an ACK is received (slow start). When cwnd >= ssthresh, it is incremented only by 1 MSS with every ACK received. If cwnd ever exceeds rwnd, only rwnd number of packets will be sent even though the congestion window may be larger.

If a loss event (timeout) occurs in the send function, ssthresh is set to half the value of cwnd at the time the loss occurred. Cwnd is then set to 1 MSS and will begin slow start again. The idea is that the new ssthresh should more

appropriately account for the current amount of network congestion than the last ssthresh and prevent the connection from transmitting too much data.

L. Bi-Directional Data Transfers

Bi-directional data transfers are made possible through multithreading. In the example of the get-post function in the File Transfer Application, two threads are created, one for uploading (sending a file from client to server) and downloading (sending a file from server to client). Both threads are started one after the other and joined. The threads both use the same client and server RTPSockets for the bi-directional transfer, but they use different RTPConnections (demultiplexing as explained above). The program continues after both threads finish.

II. RTP header structure and header fields

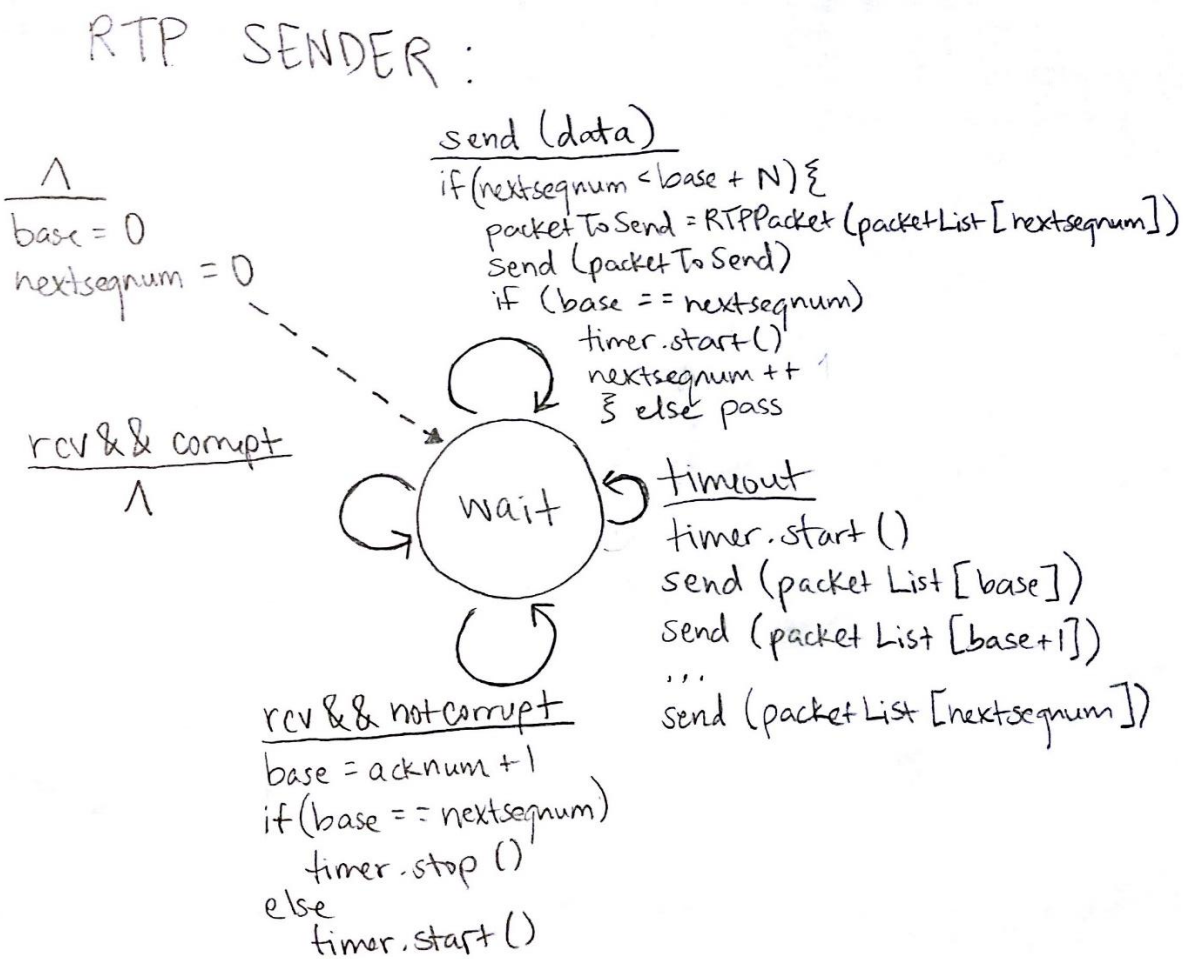
The RTP header consists of the following fields:

Field	Type	Size	Description
len_data	integer	2	Length of packet data
source_port	integer	2	Source port
dest_port	integer	2	Destination port
seqnum	integer	4	Sequence number
acknum	integer	4	Acknowledgement number
ACK	integer	1	1 if ACK, 0 otherwise
SYN	integer	1	1 if SYN, 0 otherwise
FIN	integer	1	1 if FIN, 0 otherwise
rwnd	integer	2	Receive window size
checksum	integer	4	Checksum value
eom	integer	1	1 if end of message, 0 otherwise

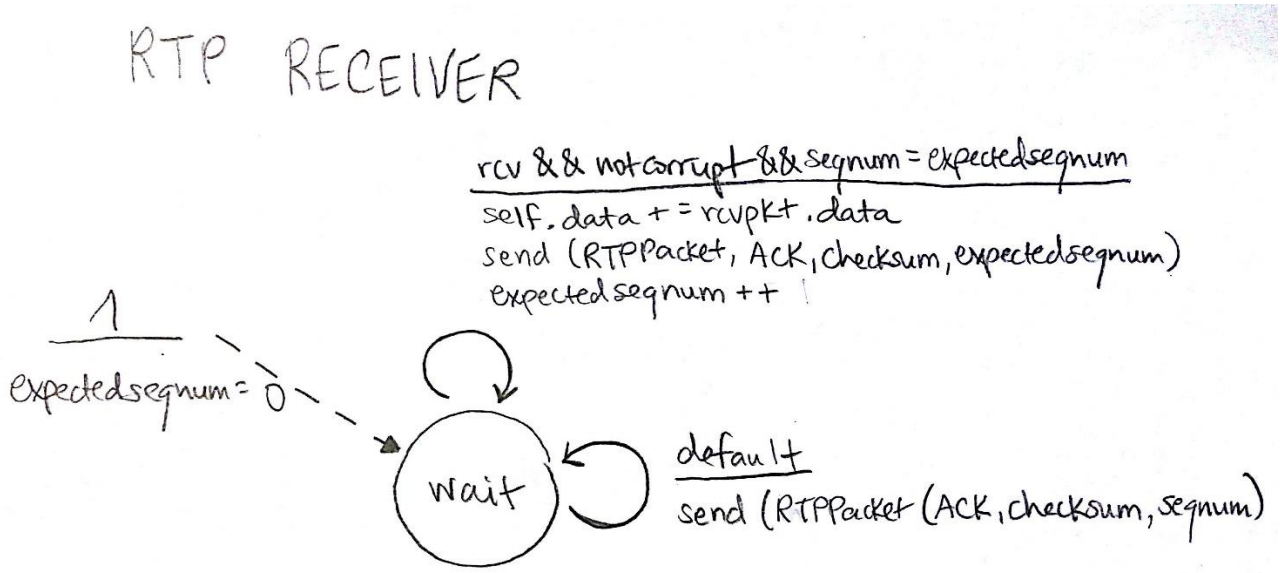
The struct library is used to "pack" the header as a binary string. The data string is concatenated onto the header before the packet is sent. When a packet is received, it is "unpacked" to retrieve the data and header fields. The "Size" column is the size of the packed field in bytes, so the total size of the RTP header is 22 bytes. Network byte order is always used.

III. Finite State Machine diagrams

A. RTP Sender



B. RTP Receiver



IV. Programming interface

A. class RTPConnection

Represents a connection over RTP

Methods defined here:

__init__(self, destination_address)

Constructs a new RTPConnection to the destination address passed in.

addData(self, data)

Adds the data passed in onto the receive buffer (self.data).

getData(self)

Returns the data from the receive buffer and resets the receive buffer.
If the receiver buffer (self.data) is empty, returns an empty string.

startConn(self)

Starts the connection by setting self.connOn to True

stopConn(self)

Stops the connection by setting self.connOn to False

B. class RTPHeader

Represents a header for an RTPPacket.

Methods defined here:

__init__(self, source_port, dest_port, seqnum, acknum, ACK, SYN, FIN, rwnd, checksum, eom)

Constructs a new RTPHeader with the fields passed in.

__str__(self)

Returns a string representation of an RTPHeader.

makeHeader(self, len_data=0)

Packs header fields and returns a byte string.

makePseudoHeader(self)

Packs certain header fields into pseudo header to be used in checksum calculation.

C. class RTPPacket

Represents an RTP packet.
MSS: maximum segment size in bytes
RTT: seconds until timer times out

Methods defined here:

__init__(self, header, data='')

Given an RTPHeader and data (optional), constructs a new RTPPacket.

__str__(self)

Returns a string representation of an RTPPacket.

getChecksum(self)
Returns the checksum (int) calculated for pseudo header and data of a packet.
See: http://www.binarytides.com/raw-socket-programming-in-python-linux/

makeBytes(self)
Returns the RTPPacket as a byte string.
len_data (int) will be used when client/server receives a packet so the length of the data is known.
See: http://docs.python.org/2/library/struct.html

setChecksum(self)
Sets the checksum field in the RTPPacket header to the value calculated in getChecksum()

Data and other attributes defined here:
MSS = 512
RTT = 2

D. class RTPSocket
Represents a socket over RTP

Methods defined here:
__init__(self)
Constructs a new RTPConnection.

__str__(self)
Returns a string representation of an RTPSocket.

accept(self)
Server side of 3 way handshake; accepts connection to client.
Returns the RTPConnection and destination address as a tuple.
If no SYN bits are received, return a tuple of two empty strings.

bind(self, socket_addr)
Binds the RTPSocket to the address passed in and starts recv thread.

clientClose(self, conn)
Closes the RTP socket and connection from the client side.

connect(self, destination_address)
Connects to the destination address passed in.
Returns an RTPConnection.

getData(self, conn_id)
Returns the data from the connection with the id passed in.

getPacket(self, bytes)
Takes in a byte string and parses it into header and data.
Returns an RTPPacket.

recv(self)

Receives data at a socket and places it in the appropriate buffer/hash map/queue.

send(self, data, addr)
Sends data through a socket to an address
data: data to send to address
address: tuple (host, port)

sendACK(self, srcport, dstaddr, seqnum, acknum)
Sends an ACK packet with scrport, seqnum, acknum to dstaddr.

sendFIN(self, srcport, dstaddr, seqnum, acknum)
Sends a FIN packet with srcport, seqnum, acknum to dstaddr

sendSYN(self, srcport, dstaddr, seqnum)
Sends a SYN packet with srcport, seqnum to dstaddr.

sendSYNACK(self, srcport, dstaddr, seqnum, acknum)
Sends a SYNACK packet with scrport, seqnum, acknum to dstaddr

serverClose(self, conn)
Closes the RTP connection passed in from the server side.

setTimeout(self)
Sets socket timeout to 2 seconds.

timeout(self, addr)
Retransmits packets from base to nextseqnum.
addr: tuple (host, port)

V. Algorithmic descriptions of non-trivial RTP functions

A. RTPConnection functions

1. __init__(self, destination_address): constructs a new RTPConnection. Sets destination host and port. Initializes variables used for flow control and congestion control.

2. getData(self): returns data from the receive buffer and resets the receive buffer.

3. addData(self, data): adds data to the receive buffer.

B. RTPSocket functions

1. __init__(self): constructs a new RTPSocket. Initializes queues, hash maps, and UDP socket. Creates (but does not start) a daemon thread that continuously calls recv.

2. bind(self, socket_addr): binds the RTPSocket to the host and port passed in. Sets self.socket_host and self.socket_port. Starts receiving thread.

3. accept(self): waits to receive a SYN, then accepts a connection to a client and sends a SYNACK. Returns a new RTPConnection.

4. connect(self, destination_address): connects to the server at the destination address passed in by sending a SYN. Starts the receiving thread for the client side. When the SYNACK is received, sends an ACK back and creates a new RTPConnection and adds it to self.connections. Starts the RTPConnection and then returns it.

5. send(self, data, addr): the data passed in is broken into segments of size MSS bytes which are made into packets and placed in a packetList. The base and nextseqnum are set to 0. The send window size self.N is set to the floor of the minimum of the receive window of the connection to send to and the congestion window of the sender. The packets in the cwnd are all sent and a timer is started. The sender then waits for a cumulative ACK. When the ACK is received, the packets up to the acknum - 1 are marked as isACKED = True.

6. timeout(self, addr): If a timer in the send function times out, the packets in the send window starting with base are re-sent.

7. getData(self, conn_id): Calls RTPConnection.getData for the connection with the ID passed in. Returns the data. If there is no data, a blank string is returned.

8. recv(self): this function is called continuously inside a receive thread. Whenever a response is received from recvfrom, the packet is examined to see if it is a SYN, SYNACK, ACK, FIN, FINACK, or data packet and whether it is corrupt (checked by calculating checksum). The packet is placed in the appropriate queue/hash map if it is not a data packet. Otherwise, if the sequence number is equal to the expected sequence number the data is added to the receive buffer of the receiving connection. The receive window size for that connection is adjusted, and an ACK is sent to the sender. If the sequence number is not equal to the expected sequence number, the ACK for the last properly received packet is re-sent.

9. clientClose(self, conn): closes the connection passed in on the client side. Sends a FIN to the server, then receives a FINACK. If the FINACK is not received, the FIN is re-sent until a FINACK is received. Then another FIN is received from the server. Sends a final ACK and waits 2 seconds to make sure the ACK gets received, then closes the UDP socket.

10. serverClose(self, conn): closes the connection on the server side. Waits to receive a FIN, then sends an ACK to the client, followed by a FIN with sequence number self.close_seq. After the client receives that FIN and sends a FINACK, the server receives the FINACK and sends one more FIN to the client. The connection.isOff is set to True for the connection passed in, and that connection is also removed from self.connections.

11. getPacket(self, bytes): takes in a byte string and parses it into header and data, then returns an RTPPacket with the header and data.

C. RTPPacket functions

1. __init__(self, header, data): constructs a new RTPPacket with the header (RTPHeader object) and data (string) passed in. Initializes self.isACKED to False.

2. makeBytes(self): packs the packet and header into a byte string and returns the byte string.

3. getChecksum(self): calculates a checksum on the packet pseudoheader and data, and returns the calculated checksum.

4. setChecksum(self): sets the checksum field in the RTPPacket header to the value of self.getChecksum().

D. RTPHeader functions

1. __init__(self, source_port, dest_port, seqnum, acknum, ACK, SYN, FIN, rwnd, checksum, eom): constructs an RTPHeader and sets header fields.

2. makeHeader(self, len_data): packs header fields into a byte string and returns the byte string.

3. makePseudoHeader(self): packs most header fields (except checksum) into a byte string and returns the byte string.

KNOWN BUGS AND LIMITATIONS

Unable to perform get-post function from FTA from two different clients simultaneously.