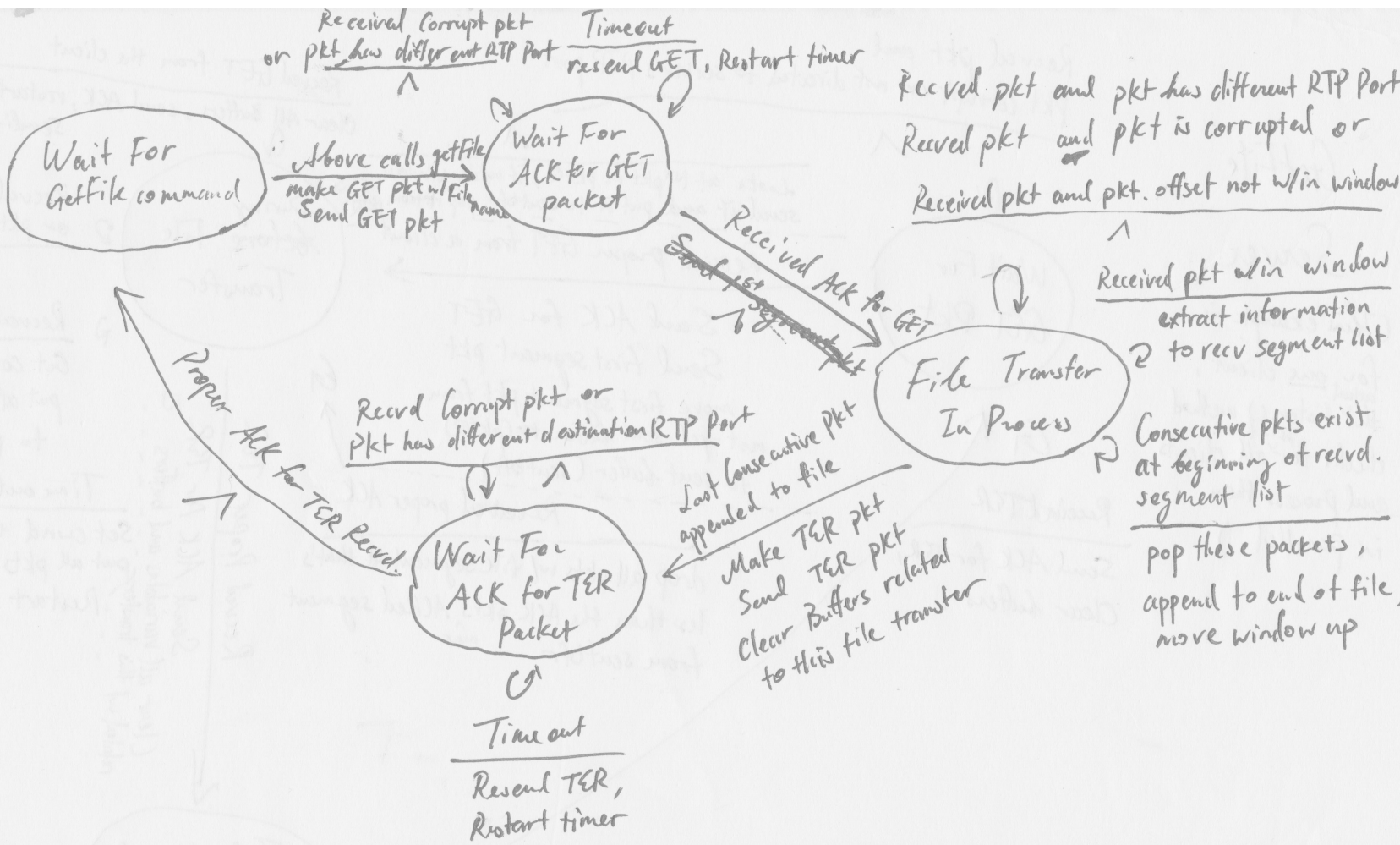
RTP Protocol Design Documentation

Cameron Gallahue and Bill Feng

1) High Level Analysis
   a) In order to send data via RTP, several pieces of information must be known. We must know where to send the data (IP address and port number), what data to send, and how much data can be sent at once (the window size in packets). The actual functionality of RTP runs as follows:
      i) The server is set up to listen for incoming packets on a particular port number.
      ii) The client sends a connection request packet to the server, containing the client's IP and port number, as well as the specific data requested.
      iii) The server accepts the connection, and divides up the requested data into packets.
      iv) All of these packets are added into the queue, and as many are sent at once as the maximum window size allows. The client acknowledges each packet as it is received, and the server shifts the sender window over to continue sending more packets in its queue. As each packet has its own destination IP and port number, the server can service more than one connection at a time by simply adding the packets for each new request to the end of the sending queue.
      v) The client then closes the connection, which sends a message to the server saying that it wants to close the connection. The server then sends an acknowledgement of the connection closing, and closes the connection. The client then closes the connection once this packet is received.
   b) RTP Header Structure
      i) ip_src: the source IP address
      ii) sPort: the RTP source port number
      iii) ip_dest: the destination IP address
      iv) dPort: the RTP destination port number
      v) sPort_udp: the source port number used by UDP
      vi) dPort_udp: the destination port number used by UDP
      vii) seqn: the sequence number of the packet
      viii) ackn: the ACK number of the packet
      ix) SYN: Boolean representing initiation of a connection
      x) ACK: Boolean representing the acknowledgement of a packet
      xi) BEG: Boolean, used to indicate the first packet of a file being transferred
      xii) FIN: Boolean, used to indicate closing a connection or end of file
      xiii) GET: Boolean used when the payload is a request for data
      xiv) POS: Boolean used when the payload is the client posting a file to the server
      xv) TER: Boolean used for notifying the other host that file transfer has completed
      xvi) Timestamp: the current timestamp of the packet used to determine timeout
      xvii) RWND: receive buffer window for the sender host of the packet
      xviii) Checksum: the checksum value of the packet
      xix) Length: the total length of the packet
   c) Finite State Machine Diagrams (See Next Page). The getFile client and server FSM are shown. postFile client FSM is extremely similar with getFile server FSM, with the addition of a name packet. For single packet transfer (database queries), a simple wait-for-ack procedure is used, if ACK is not received within a certain time, then the packet is resent.

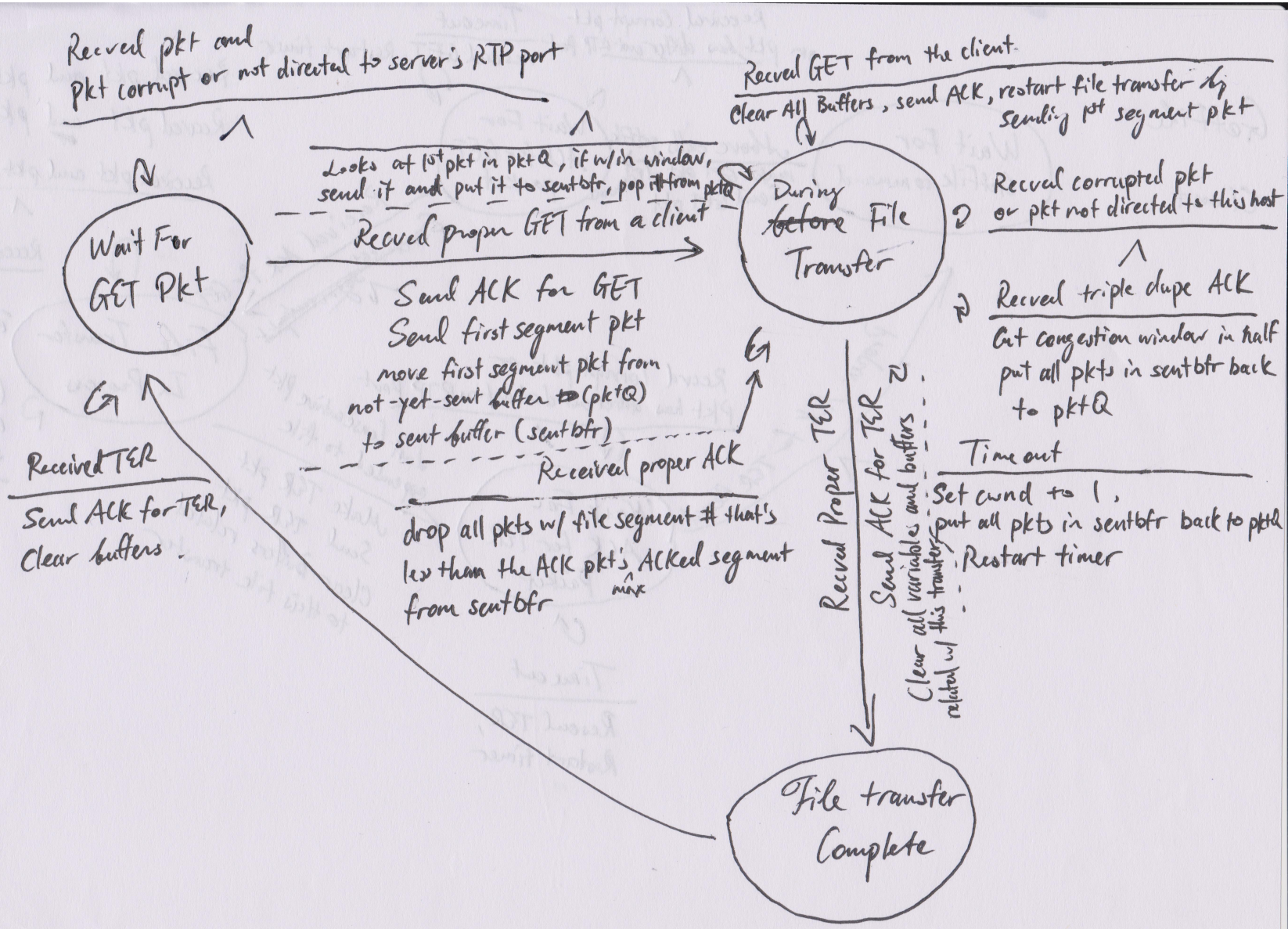
GetFile:

Client



GetFile Server:

(This example is for one client, the actual listen() method checks for all clients' pkts from and process them in parallel.)



d) RTP API
   i) __init__(ip_addr, udp_port, rtp_port, server, receiveWindow):
      (1) ip_addr is the IP address to be used, either the IP of the server if used client-side, or the local address (127.0.0.1) if used server-side
      (2) udp_port is the UDP port number of the server. Should be whatever number the server listens on.
      (3) rtp_port is the RTP port number used to number the particular connection. This allows the connection to be multiplexed, sending to the same physical port, but separating the data at the source by its "virtual" port number, or its connection number.
      (4) server is a Boolean that is set to true if this is created server-side, or false from the client side.
      (5) receiveWindow is the size of the receiver window in packets. RTP will allow this number of in-transit packets at any given time.
      (6) This method is the constructor method for the RTP class
   ii) connect(ip_client, uPort, dPort):
      (1) This method is only called client-side.
      (2) ip_client is the IP address of the local machine (the client, not the server)
      (3) uPort is the physical UDP port to send the data back to
      (4) dPort is the RTP port to send the data back to
   iii) close(ip_dest, uPort, dPort):
      (1) ip_client is the IP address of the local machine (the client, not the server)
      (2) uPort is the physical UDP port to send the data back to
      (3) dPort is the RTP port to send the data back to
   iv) listen():
      (1) This is called in an infinite loop server-side, and automatically handles all traffic internally, accounting for all file sending and posting, as well as establishing connections and closing connections with a client.
      (2) It returns *None* if the packets received are internal packets (for establishing a connection, closing a connection, ACK for a packet, or for transferring a file), and returns the packet otherwise.
   v) getPost(getName, postName, ip_dest, uPort, dPort):
      (1) This is used for synchronous get-post from and to a host.
      (2) getName is the file to get from server, postName is the file to post on the server, ip_dest, uPort, and dPort are the IP address, UDP port, and RTP ports of the server.
      (3) This method is called client-side only.
   vi) getFile(filename, ip_client, uPort, dPort):
      (1) This is used for requesting a file from a server
      (2) Filename is the name of the file to be retrieved
      (3) ip_client is the IP address of the local machine (the client, not the server)
      (4) uPort is the physical UDP port to send the data back to
      (5) dPort is the RTP port to send the data back to
   vii) sendFile (filename, ip_client, uPort, dPort):

(1) Filename is the name of the file to be sent
(2) ip_client is the IP address of the local machine (the client, not the server)
(3) uPort is the physical UDP port to send the data back to
(4) dPort is the RTP port to send the data back to
(5) This method does not send the file directly. Instead, it decomposes file into packets, and queue the packets to the send buffer ("pktQ" in the program).

viii) send(message, ip_dest, uPort, dPort):
(1) message is the byte array of data to be sent to the recipient
(2) ip_client is the IP address of the local machine (the client, not the server)
(3) uPort is the physical UDP port to send the data back to
(4) dPort is the RTP port to send the data back to
(5) This method sends a single packet to the destination hosts, it checks for ACK to ensure that the packet is received by the other host, else it resends the packet.

ix) recv(ip_dest, uPort, dPort):
(1) ip_client is the IP address of the local machine (the client, not the server)
(2) uPort is the physical UDP port to send the data back to
(3) dPort is the RTP port to send the data back to

e) Algorithms of Non-Trivial Methods:
- For file transfer, the sending side packets are split to 3 chucks: sent and ACKed consecutively, sent and not ACKed consecutively, and not sent. Packets to be sent are placed in packet queue (pktQ), while those sent and not ACKed consecutively are placed in send buffer (sentbfr). The program continuously checks for ACKs, and performs similar functions as the TCP when triple duplicate ACK or a timeout occurs.
- Similar to TCP, the ACK packets represents the highest sequence number ("pkt.hdr.offset" in the program) of consecutively received packets.

f) Other Information

i) How does RTP perform connection establishment and connection termination?
(1) Connections are established by sending a SYN packet to the server, which is responded to by a SYNACK. Once the client receives the SYNACK, it will be able to send its request to the server. To terminate the connection, the client sends a FIN packet to the server, which responds with a FINACK. Once the client receives the FINACK, the connection is closed.

ii) How does RTP perform window-based flow control?
(1) The RTP takes in a maximum window size on creation. The window size is maintained by allowing the first n packets in the queue to be sent given that n is the window size. Once the first packet in the queue is ACKed, it is removed from the queue, allowing the next packets in line to be sent.
(2) The maximum window size is the minimum of the Congestion Window and the destination's Receive Window. Congestion Window is dynamically scheduled to implement congestion control.

iii) How does RTP detect and deal with duplicate packets?
(1) For window size 1, the duplicate packet is ignored because the *send()* method returns before the ACK for the duplicate packet arrives. If host A sends a packet to host B and the packet duplicates, host A will expect an ACK from host B, and will return from the *recv()* method immediately after an ACK is received. Host B will still send an ACK for both packets, but the second ACK will not be captured.

(2) For larger window sizes (which is during file transfer in the context of this assignment), there is no mechanism to detect and drop duplicate packets. Instead, the RTP protocol maps each packet within the congestion window to its file sequence number (pkt.hdr.offset). When a duplicate packet is received, it simply replaces the original packet in the dictionary, which is entirely harmless and would not consume extra time.

iv) How does RTP detect and deal with corrupted packets?
   (1) When a packet is corrupt, its actual checksum would not match the checksum value in its header. If there is a mismatch, RTP pretends as if this packet is never received. Therefore, any corrupted packet will be logically treated as a lost packet.

v) How does RTP detect and deal with lost packets?
   (1) For window size of 1 (during connection establishment, connection closing, and single packets communication), if an ACK for a sent packet is not received within 2 seconds, the sender will resend the packet and reset timer.
   (2) For window size greater than 1 (during file transfer), similar with TCP, lost packets will result in either a triple duplicate ACK or a timeout (as the ACK packet's *hdr.offset* indicates the highest consecutively received file sequence number + 1). Upon both situations, the sender will move all packets in the sent packet buffer whose file sequence number are greater than the maximum ACKed file sequence number to the not-yet-sent-packets buffer. Also it cuts congestion window by half (with a minimum of 1) upon a triple duplicate ACK, and cuts congestion window to 1 upon a timeout. Timeout is detected by iterating through the sent packets buffer and check if current time is more than 2 seconds after the most recently sent packet.

vi) How does RTP detect and deal with re-ordered packets?
   (1) Similar to duplicate packets, RTP does not try to deliberately separate re-ordered packets from original ones. For window size of 1, the receiver side will perform the identical action, sending an ACK with ack number corresponding to the re-ordered packet's sequence number. For larger window size during file transfer, they will be checked for whether their file sequence number is within the current congestion window.

vii) How does RTP (de)-multiplex data to different RTP connections at the same host?
   (1) RTP uses both the physical port number (used for UDP messages) as well as a "virtual" port (rtp_port) that's used to determine what particular connection a particular packet is intended for. This way, a single UDP port can be used to service multiple information streams at once.
   (2) If the RTP port destination of a packet does not match the RTP port of the receiver, it will ignore the packet.

viii) How does RTP support bi-directional data transfers?
   (1) Both the client and server handle outgoing data and incoming ACKs (or vice versa) already, so during bi-directional data transfer, the RTP header is checked to see what type of packet is being sent, and handles it whether it is a data packet or an ACK.
   (2) In the get-post scenario, packets that contain actual data does not have the ACK flag set, while ACK packets will always contain no data and have the ACK flag set. Every single iteration (*listen()* method for the server-side and *getPost()* method for the client) is consisted of a sending phase and a receiving phase, and one packet is sent and received each call to make sure that get and post operations are synchronous.

ix) How does RTP provide byte-stream semantics?

(1) Byte-streaming is supported as the message to be sent is supplied as a byte array, which is split into payloads to be sent one after the other. As the requirement is for each packet to be less than 1000 bytes, the maximum length of a segment is limited to 900 bytes.

x) Are there any special values or parameters in your design (such as minimum packet size)?
   (1) Congestion window is originally set to 1 for file transfer, and will increase when there is no transfer failure or timeouts, similar to TCP.
   (2) The TER flag is used for informing the destination host that the file transfer is complete and every segment of the file has been received, upon receiving a TER packet, the host will clear all buffers related to the file transfer and send an ACK to the TER packet.
   (3) In file posting, the BEG flag is set only for the packet containing the file name when posting a file, and is not set for the actual first segment of the file.