RTP Design Docs

Luka Antolic-Soban & Tre Cooper | Constantine Dovrolis, CS 3251 | Design and Implement a RTP

This is our Design Documentation that will provide you the necessary information to understand our protocol.

Table of Contents

How does RTP work?

Our RTP design works in a similar fashion to the infamous TCP

The purpose of this protocol is to provide a reliable connection service between two pairs of processes. The first is basic data transfer. RTP is able to transfer a continuous stream of data in each direction between its users by packaging a certain number of bytes into packets for transmission through the internet. RTP decides when to block or forward data on its own.

Reliability:

RTP has to recover data that has been damaged, lost, duplicated or delivered out of order by the communication system. The way this is achieved is by assigning a sequence number to each packet transmitted and requiring a positive acknowledgement of said packet. If the ACK is not received in a certain amount of time interval, then the data is retransmitted. On the receiver side, the sequence numbers are used to correctly order the packets that may have been received out of order and to get rid of duplicates. The way this works is by first holding the packets in the window in their respective sequence. They are stored in a HashMap based on this sequence and we iterate through it, looking for gaps in the keys. These gaps are the packets that haven't arrived yet. Corruption of the packet is also handled by adding a checksum to each segment transmitted, checking it at the receiver and getting rid of damaged packets. As long both sides of the connection are functioning properly, no transmission errors will occur.

Flow Control:

RTP gives the receiver the duty to decide the amount of data sent by the sender. This is done by returning a window with every ACK indicating that a range of acceptable sequence numbers past the last packet is successfully received. This window indicates an allowed number for packets that the sender can transmit before getting permission to do so.

Multiplexing:

For our RTP to allow many processes within a single host to use RTP communication at the same time, RTP gives a set of ports within each host. Since RTP is working on top of UDP, we use the UDP addresses and ports in conjunction with our own RTP port system to uniquely identify each connection. A pair of RTP sockets is this identifier as well. This binding of a socket is handled by each host.

The Connections:

The reliability and flow control methodologies that are described require that RTP load and maintain status information. Therefore, overall, with the combination of RTP sockets, sequence numbers, and window sizes, a connection is born. For two processes to begin communication, RTP must first establish a connection. Since these can be established between unreliable hosts a 3 way handshake system takes place.

Operations:

Each process sends data by calling on the RTP socket passing bytes of data as parameters. RTP takes this data and packages it into “packets” and then sends them to a sending queue so that the UDP socket may handle the sending along the network. The receiving UDP socket takes the data from a packet and places it into a receive queue for that particular port. The socket corresponding to that port number then takes the data from the queue and places it into the receiving buffer to pass it to the user. RTP uses control information in the packets to ensure reliable data transmission.

Reliable Communication:

A byte stream of data that is sent on an RTP connection is sent reliably and in order at the destination. As stated before, it is made reliable by the use of sequence numbers and acknowledgements. Each packet is given a sequence number before it is sent. Packets also carry along with them an acknowledgement number which is the sequence number of the previously received packet. When RTP sends a data packet, it is kept in a list, where a timer is set the moment the packet leaves. If an acknowledgement is not received before the timer runs out, then the packet is sent again. To dictate the flow of data between RTP sockets, a flow control mechanism is used. The RTP receiving end shares a window to the sending RTP and is the number of packets, starting with the Acknowledgement number, that the receiving RTP is prepared to receive. If RTP receives a packet with the same sequence number of a previous acknowledge packet it simply discards it if it is outside of the window or replaces it with the one that is already in the current window. Whenever a packet is delivered, RTP immediately checks the checksum of the packet. It takes the byte array and calculates it with the header number. If it matches, it continues to the recipient or the packet gets drops if the checksum isn’t a match.

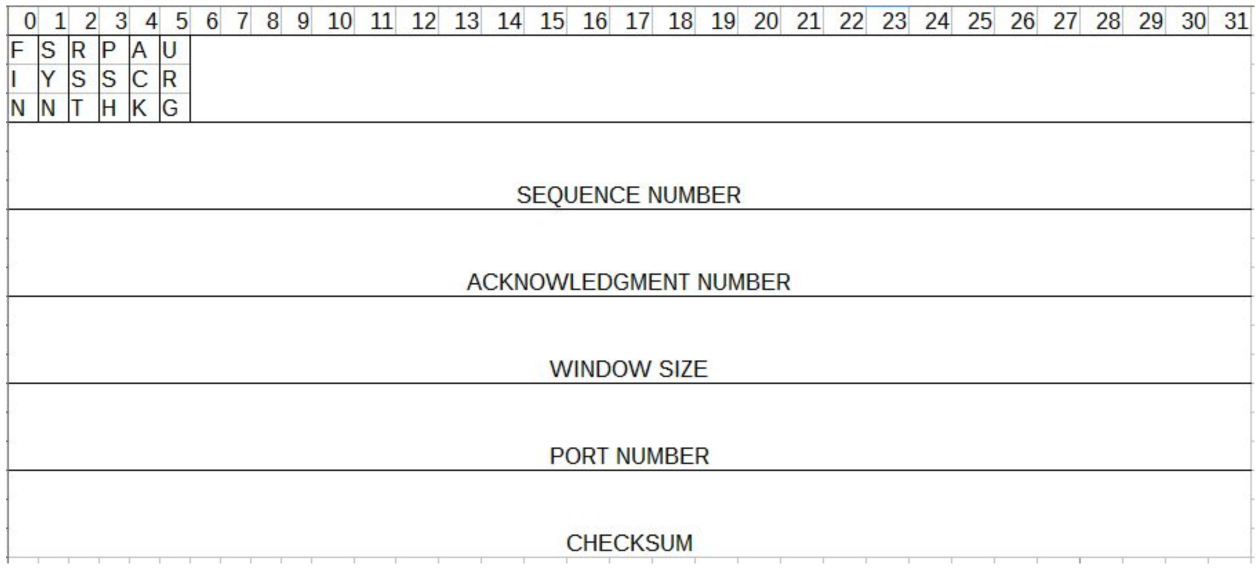
Establishing a connection:

RTP, when identifying different streams, uses a unique port number which is then shared amongst connections. A connection is given by the pair of sockets. A local socket or RTPStack as we have defined, can engage in many connections at once. This data can be sent in both directions. The socket that wants to connect to another one initiates the call by the ip and port. This method establishes connections by sending a SYN control flag to the receiving end. This starts a process of a 3-way handshake. The opposite it also done when trying to close a connection but is instead initiated with a FIN control flag.

Congestion Control:

None

Header Format



Flags: 32 bits

Most of these are not used. They are kept in their for further implementation. Only SYN, FIN, ACK, and RST are used, which correspond to synchronize, finish, acknowledgment, and reset.

Sequence number: 32 bits

The sequence number of the first data array in this packet except when a SYN is present. If the syn is present, then the sequence number is the initial sequence number.

Acknowledgment number: 32 bits

If the ACK control flag bit is set then this field assumes the value of the sequence number that we have already received. Once established with a connection, this is always sent.

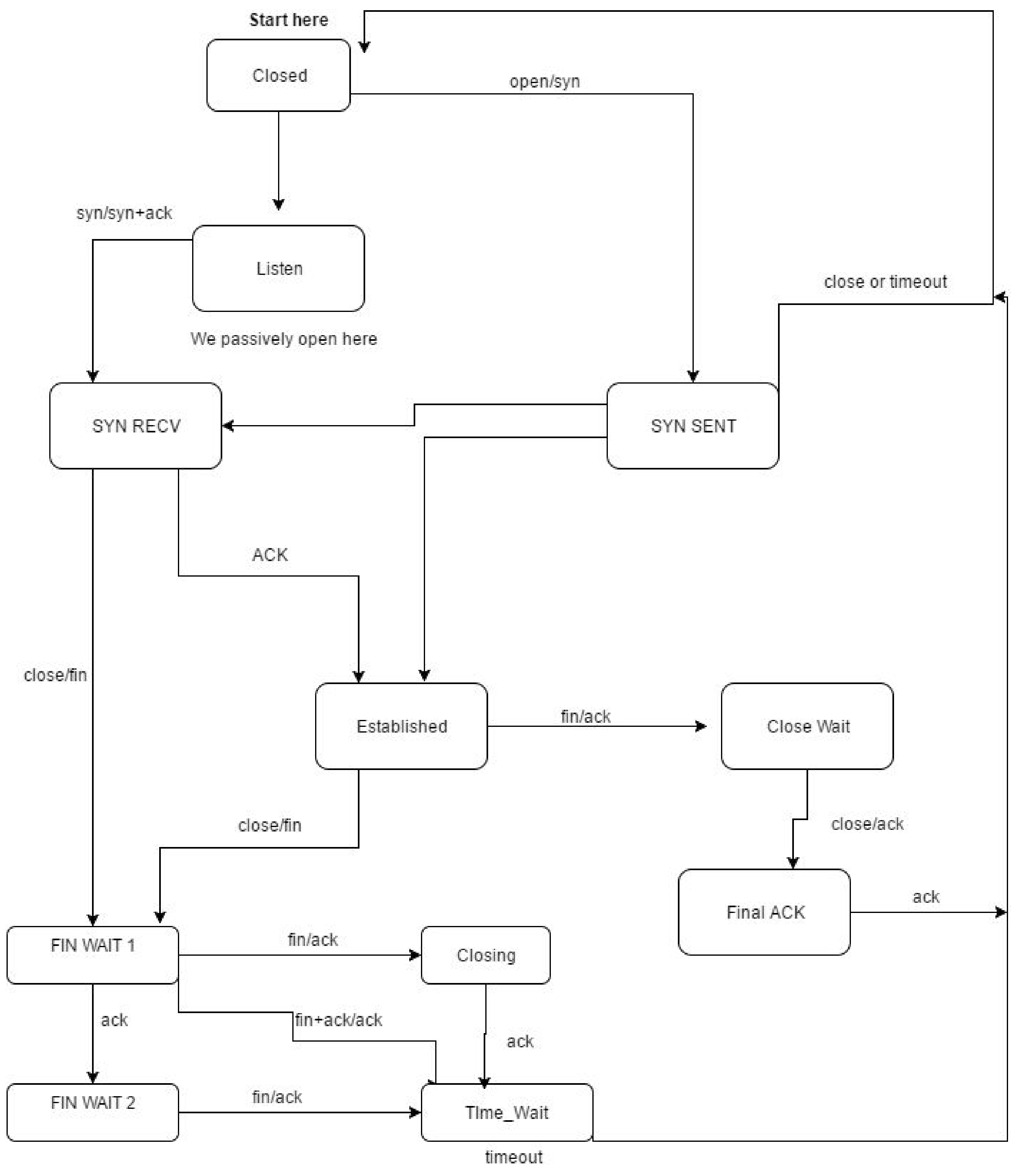
Window Size: 32 bits

This is the number of data segments starting with the one given by the acknowledgment field which the sender is accepting.

Checksum: 64 bits

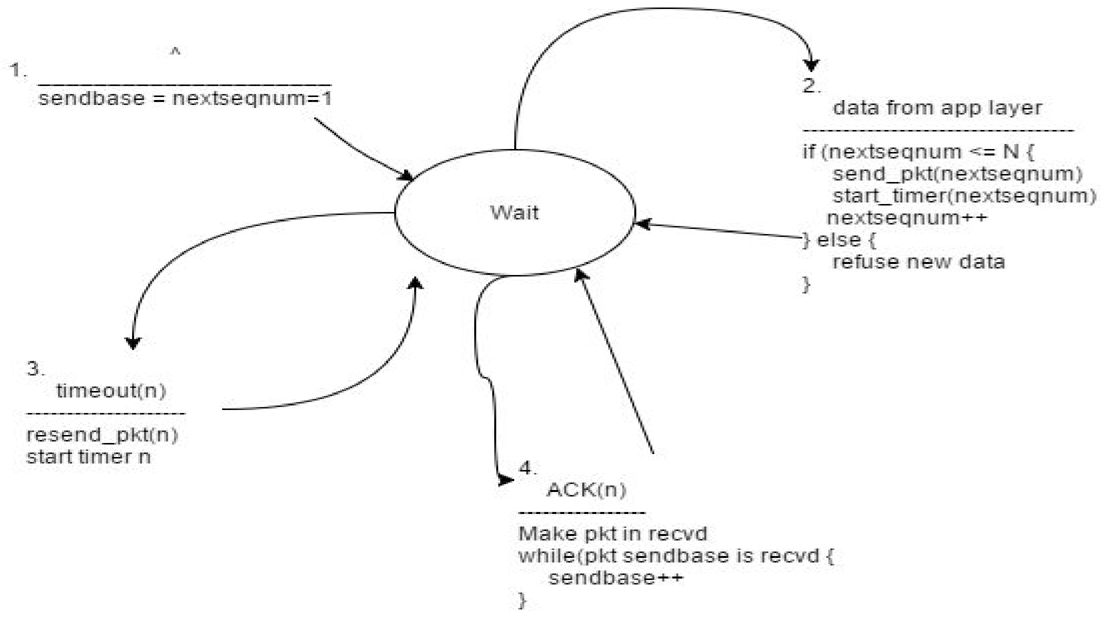
This is the field where the CRC32 checksum is made by passing in the byte array of the packet with a checksum of 0 and then places the calculated checksum inside this field.

RTP STATE MACHINE



RTP ESTABLISHED STATE FOR BOTH RECEIVER AND SENDER END POINTS

SENDER



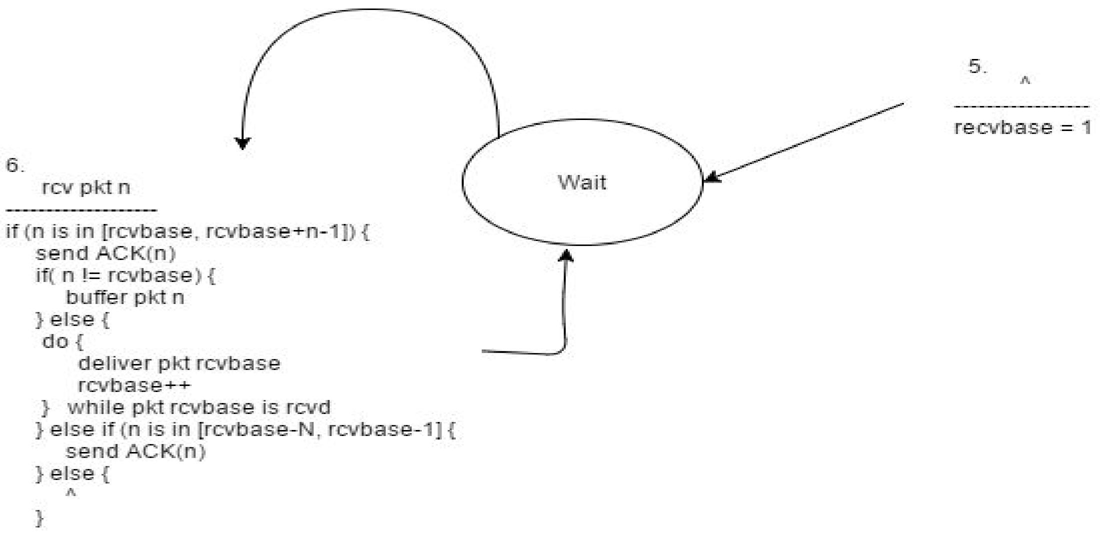
```
1.          ^
  ________________________
  sendbase = nextseqnum=1
```

```
2.
    data from app layer
-----------------------------------
if (nextseqnum <= N {
    send_pkt(nextseqnum)
    start_timer(nextseqnum)
   nextseqnum++
} else {
    refuse new data
}
```

```
3.
    timeout(n)
--------------------
resend_pkt(n)
start timer n
```

```
4.
    ACK(n)
----------------
Make pkt in recvd
while(pkt sendbase is recvd {
    sendbase++
}
```

RECEIVER



```
5.    ^
-----------------
recvbase = 1
```

```
6.
    rcv pkt n
-------------------
if (n is in [rcvbase, rcvbase+n-1]) {
    send ACK(n)
    if( n != rcvbase) {
        buffer pkt n
    } else {
     do {
         deliver pkt rcvbase
         rcvbase++
     }   while pkt rcvbase is rcvd
    } else if (n is in [rcvbase-N, rcvbase-1] {
        send ACK(n)
    } else {
        ^
    }
```

RTP API

Class RTPStack

Constructor:

RTPStack()

Creates an un-initialized RTPStack, an object that contains all of the data structures necessary to create a listening UDP socket.

Methods:

void **init**(InetAddress bindAddr, int port) throws SocketException;

Initializes the Datagram socket bound to a specified local address. The local port must be between 0 and 65535 inclusive. If the IP address is 0.0.0.0 the socket will be bound to the wildcard address. Data structures and threads for sending and receiving on that particular address + port will then be spawned.

This method throws a SocketException - if the socket could not be opened or the socket could not bind to the specified local port

Class RTPSocket

Constructor:

RTPSocket(InetAddress bindAddr, int port) throws IOException;

Creates a new RTPSocket to which a connection may be paired to.

Parameters- the port number and local address that the socket will bind to.

This method will throw an IOException if an I/O error occurs while opening the Datagram socket.

Methods:

void **connect**(InetAddress serverIP, int serverPort) throws IOException, InterruptedException;

Connects a socket to a remote host on a given port. This method begins the three-way handshake and perform RTP negotiation to open a connection. It sends a SYN, waits for a SYN-ACK and then sends an ACK. If this is all good then it returns from the function and the connection is established.

Method throws an IOException or InterruptedException if the socket is already closed or if the threads interrupt this method.

void **accept**() throws IOException, InterruptedException;

Listens for a new socket connection to bind to. It uses the RTPStack as an interface to grab any new packets that comes from an unknown source. This method takes the first packet from the unestablished queue for the RTPStack and begins the three-way handshake with the remote host. It receives a SYN, sends a SYN-ACK and then receives an ACK before returning.

Parameters are none.

Method throws an IOException or InterruptedException if the socket is already closed or if the threads interrupt this method.

int **receive**(byte[] usrBuf, int usrOff, int usrLen) throws IOException, InterruptedException;

Reads up to a usrLen bytes of data from the stream into an array of bytes. An attempt is made to read as many as usrLen bytes but a smaller number may be read. The number of bytes actually read is returned as an integer.

This method blocks until input data is available, end of file is reached, or exception is thrown.

If usrLen is 0 then no bytes are read. There is an attempt to read one byte. The first bytes read is stored into the offset and so on. You may call this method multiple times in succession. If the first such call results in an IOException, that exception is returned from the call to the read(b, off, len) method.

Parameters are the usrBuf - buffer in which to read

usrOff - that start offset in array b at which the data is written

usrLen - the max number of bytes to read

Throws an IOException - If the first byte cannot be read for any reason other than end of file, or if the socket has been closed, or if some other I/O error occurs.

void **send**(byte[] data) throws IOException, InterruptedException;

Sends data.length bytes from the specified byte array to this output stream. It will block until all the data is sent and all the acks and been received for the data.

Parameters are data - the byte array to send

This method throws an IOException if an I/O error occurs. In particular, an IOException is thrown if the socket is closed.

void **close**() throws InterruptedException;

Frees a socket, disconnecting it from the remote host that is was connected to if need be. If it needs to disconnect from the host first, it prepares a FIN packet to send to the end-host, waits for a FIN-ACK, and finally closes the connection if the ACK is delivered successfully. Once the socket has been closed, it is not available for further use. A new socket must be created.

Parameters are none

This method throws an InterruptedException error if it is trying to close on an already closed socket, which may be the case if the end-host has initiated a close before it did.

void **setWindowSize**(int size) throws Exception;

Sets the window size for this socket to the specified amount. Increasing this can increase the performance of the socket while decreasing may reduce the receive queue of incoming data. This value is also used in conjunction with the receive buffer size.

Parameters size- the size to which to set the window size. This values must be greater than 0

This method throws an exception if the values is less than or equal to 0. It also throws an exception if there is an error in the underlying RTP protocol, such as an already closed socket/connection.

Algorithmic Descriptions of complex RTP functions

Close

Closing a connection may begin from either host. When the initiator calls this function, it spawns a close state listener and sends a FIN flag to the end host. This thread waits for the ACK to the FIN-ACK by using the local machine time and a timestamp whenever a packet was sent in the three-way teardown. It was important to give the thread the capability to timeout due to the fact that the socket continues to act as if the connection is still established and only when this thread notifies the socket, does it close. We decided to abstract the close thread away from the socket so that all the other socket methods did not have to support being closed in the middle of their operation.

makeIntoPacket / toByteForm

This function serves the purpose of interpreting a RTP packet from a buffer of bytes. As we receive a new DatagramPacket from our underlying DatagramSocket, we read in the data buffer in total. Due to our set header fields we now loop through each byte and can assign the byte values appropriately to the class variables. Java has set the values of any integer in a 4 byte form. All of our header fields are 32 bit values so we loop through the buffer in 4 byte segments. These segments each correspond to a section of the header field. The last byte at the end of the header to the end of the stream is the data of the packet. The functionality works the same for both function, just in the opposite direction.

Receive

This method waits for a stream of bytes to return to the user at the application level.

The method starts by iteratively receiving packets from the receive queue for the particular connection (Waiting for a packet to arrive). For each packet received we first check the sequence number and flags to make sure that this is indeed a data packet. If it is, we check the sequence number and compare it to the base of our sliding window (seq_num > base && seq_num < base+slidingWdnSize) of the receive buffer. We then immediately create an ACK packet and send it to the UDP socket. If they are equal to each other, then that packet and all subsequent packets (no gaps) are compiled for the application layer in the receive buffer as payload. If the packet falls in the range of this window, we simply store this packet in the sliding window. The packet is dropped if those two conditions are not satisfied. If and when the receive buffer is full of payload, data is passed up to the application layer, reserving all method variables for consecutive calls to receive. In any event that the packet is flagged as the last data packet, all of the data in the receive buffer is passed to the application layer, signaling the end of the receive call. Due to the capacity of the receive buffer, it is likely called in a while loop for large byte stream readings.

Send

This method sends a stream of bytes to the end-host, maintaining its own Acknowledgments.

This method starts by iteratively breaking down the byte buffer into the number of necessary packets (1000 bytes max packet size). For each packet a sequence number is generated and placed inside of the RTP header. The last packet to send is given a special flag so that the end-host knows when this particular send is complete. These packets are stored for later distribution, ending the pre-processing phase. The next phase begins the actual sending of the packets. To start off, a full window size of packets are sent. A timelog is taken of the time when the sliding window has been sent. Proceeding this, we continuously wait for new packets to arrive, these packets should contain the ACKs for the corresponding sequence number of the transmitted packets. In the background, if the timer passes the previously stamped timelog (by a certain value), we check which packets have not been Acknowledged and retransmit those packets. The timelog is reset to give the recipient time to acknowledge the retransmissions. As this continues the window is slowly moving up until it reaches the end of the send buffer. Returns once all of the data has been ACKed.