CS 3251: Programming Assignment #3
Reliable Transport Protocol Design Report

Tanay Ponkshe (tponkshe3)
Utkarsh Garg (utkarsh6)

NOTE: Unimplemented Portions are highlighted throughout the report.

Introduction

The Reliable Protocol Implementation provided in this report is a connection oriented protocol derives its reliability elements from TCP and attempts to add efficiencies using basic forward error correction mechanisms and provides an additional non trivial checksum to maintain integrity of the information. The protocol has been designed on top of UDP.

RxP Protocol Design Specifications:

Pipelining: RxP is a pipelined protocol, implemented using sliding windows. Both the client and server maintain 2 buffers each that are initialized during the connect (handshake) phase of the protocol. The two buffers on both sides are the send buffer(contains packets it wants to send to the other side), the receive buffer(fills up with received packets). The send and receive buffers are python dictionaries mapping sequence numbers to the data received in the packet. Each element of the send and receive buffers has enough space to hold one packet. Both sides determine the next packet using the next expected ACK number sent by the sender(client/server); they also maintain a variable nextSequence which contains the next expected sequence number. Out of order packets are automatically handled as they are indexed by sequence number in the receive buffer. This sequencing allows both sides to determine missing packets in between, and request a retransmission. The window size in the pipelined RxP is designed to be configurable based on the network congestion. This means that the window size can grow and shrink based on network conditions. This feature is implemented to facilitate window-based flow control. (The current implementation has a default window size)

Lost Packets: The RxP implementation sends an ACK per packet received. The Sender keeps sent packets in its buffer until an ACK for the packet is received. As soon as the ACK is received, the packet is discarded and the buffer element frees up for new packets. The buffer is implemented as a dynamic data structure which adjusts size on deletions and doesn't lead to NULL values. There is a 2 second timeout which forces a retransmit, in the case that an ACK is not received. The 2 seconds is a starting value and will be dynamically changed using the Smoothed Round Trip Time (SRTT) implementation. The RxP header has a 16 bit field for packet sequence numbers, which allows the protocol to avoid the problems faced by the alternating bit protocol.

Every ACK contains the remaining receive buffer size, k. When the receiver buffer is full, it sends an ACK with k=0. This causes a 200ms pause in the sender side, after which the sender starts transmitting again. If the receiver is still full, It sends an ACK with the previous sequence number and k=0, again.

Note: Calculation of remaining receive buffer size k:
k = (Total buffer size - (Last Sequence number - Number of elements in missing buffer))

Corrupted Packets: The design implementation for corrupted packets includes a user choice; they can choose to opt for regular retransmission of packets in case they are corrupted as in TCP or they can opt for a forward error correction approach, explained in the the Approach 2 section below.

Approach 1: In this method, the receiver uses the BSD checksum to reject a corrupted packet and does not send the ACK. On timeout, the sender retransmits the non-acked packet. The BSD checksum implementation is discussed in the checksum section.

Approach 2: This approach uses the Parity Check Forward Error Correction technique. In particular, because of the possibility of errors in multiple bits, a 2D parity check is implemented.

The bit string of the message to be sent is arranged in a square matrix with dimensions (√size(message) + 1) * (√size(message) + 1). The extra row and extra column are for the parity bits. The implementation in this report is an even parity implementation i.e. If the number of ones in row i are odd, the parity bit is set to 1, else, it is set to 0.

When the message is received at the receiving end, the number of ones in each row and column are counted. Whenever a row or column has an odd number of ones, it is certain that there is an error. There must be a corresponding column or row with an odd number of ones. Using these indices, the exact bit is located, and flipped. This eliminates the need for retransmission of corrupted packets. The additional overhead is 2*√size(message). We later found out that this method can only correct one bit errors. We looked at other, more robust FEC implementations and an example test file has been provided, but the protocol does not use it.

Duplicate Packets: The receiver has a buffer of sequence numbers of received packets mapped to their data.and one with sequence numbers of missing packets. If an incoming packet is found out of order (using the numSequence variable described in the Pipelining section) and it is not one of the sequence numbers in the missing buffer, the packet is simply discarded, and an ACK with the packet's sequence number is sent back to the sender.

Out-of-Order Packets: As discussed earlier, the receiver has a buffer of the received packets and their sequence numbers. A variable contains the sequence number of the last packet passed up to the application layer. Every packet being passed up is checked and must be one greater than this variable. Once the packet is passed, the value of this variable is replaced with the packet's sequence number.

Bi-directional Data Transfer:
The protocol provides bi-directional data functionality by leveraging the concept of Piggybacking. This implementation is very efficient because it combines ACKs with data packets and thus allows for seamless bi-directional transfer of data. This erodes the distinction of "sender" and "receiver" because now, both machines can send and receive data.
This works in the following way:
We introduce an ACK field in the Data packet
Suppose "B" sent data to "A". Now, if machine "A" wants to send data to party "B", it will send the data along with this ACK field.
If "A" only wants to send the acknowledgment, then it sends a separate ACK frame.
If station A wants to send only the data, the ACK field is set to the last acknowledgement and is sent along with the data. Here, "B" would simply ignores this duplicate ACK frame.
The previous approach was a full duplex approach. We take a simpler approach for our implementation. According to the RxP and FxP specifications of the assignment, only either the server or the client will be sending files at a time i.e. both get and post will not be called simultaneously, only one will be called at a time. The client and server both have data sending and receiving capabilities; For this we implemented an additional leg to the initial handshake. Our protocol now has a 6 way handshake. In the last portion of the handshake, the server waits

for a command, and the client sends it either a get or a post request. The client and server decide their roles based on this request and then the file transfer proceeds.

Checksum: The RxP design in this report uses the BSD checksum, as described here (http://en.wikipedia.org/wiki/BSD_checksum). This computes a 16 bit checksum by adding up all the byte-values in the message string.
The pseudocode is as follows:

```
Checksum():
    checksum  = 0
    for line in the packet other than the checksum line:
        for ch in line:
            checksum = (checksum >> 1) + ((checksum & 1) << 31)
checksum += ch
            checksum &= 0xffffffff
    return checksum
```

Special/Extra Credit Features: Our protocol design has several special features:

1. **RSA encryption**: For each packet, the message is RSA encrypted. During the connect phase:
The receiver generates 2 large random primes p and q.
It then calculates $n = p*q$ and $f(n) = (p-1)*(q-1)$. p and q are then discarded.
A number e is selected, such that e and n are co-prime. (n,e) is then transmitted as the public key for the rest of the encryption.
$d \equiv e-1(\text{mod } f(n))$ is now the private key.
Before the sender sends a message, it encrypts it using $c \equiv me(\text{mod } n)$.
On receiving the message, the receiver decrypts using its private key, $m \equiv cd(\text{mod } n)$.
This is secure because cracking it is as hard as factoring an integer, which is NP-Hard.
The protocol is capable of rss encryption but it has not been implemented. The initial handshake involves an exchange of public keys and both sides store their private keys. Encryption and decryption however are not being performed in the current implementation.

2. **Non-trivial checksum** - we are implementing a non-trivial checksum, as described in the section above.

3. The design includes **forward error correction**, as an attempt to make the transport more efficient. The error correction algorithm is described in the section above.

4. The design has the capability for **bidirectional data transfer**, as described in the section above (both GET and POST have been implemented).

High Level Overview of RxP:

The RxP protocol described in this report takes inspiration from TCP and provides several functional enhancements in order to improve upon the basic TCP features. It implements a Pipelined design similar to Selective Repeat, which is an improvement over the traditional stop and wait protocol. With this pipelined design, the sender sends a number of frames specified by a window size, without the need to wait for individual ACK from the receiver as in go-back-n ARQ.The receiver accepts out-of-order frames and buffers them. The sender individually retransmits frames that have timed out. The receiver would keep track of the sequence number of all the missing frames and sends the number of the earliest missing frame with every ACK it sends. If a frame from the sender does not reach the receiver, the sender continues to send subsequent frames until it has emptied its Window. The receiver continues to fill its receiving window with the subsequent frames, replying each time with an ACK containing the sequence number of the earliest missing frame. Once the sender has sent all the frames in its window, it re-sends the frame number given by the ACKs, and then continues where it left off. For this to work, the sender and receiver window sizes should be the same. Also, the receiver sequentially sends the sequence numbers of the missing packets in subsequent ACK.

Connection establishment is a 6 way handshake and connection termination is a 4 way handshake in RxP. This is exemplified by the packet flow diagram below.

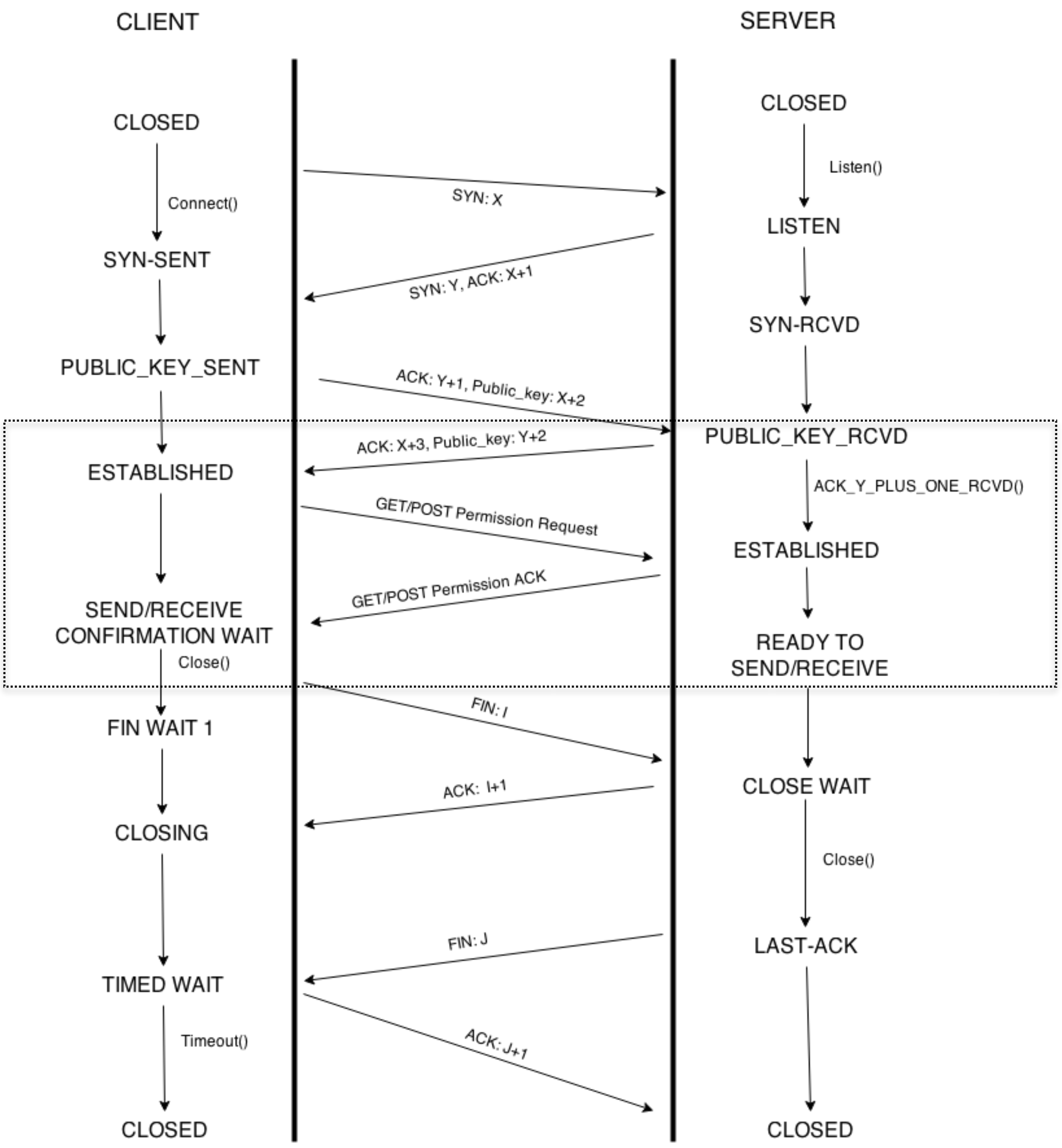
The connection establishment also entails the exchange of public keys and the information about the type of request the client is making (GET/POST). The connection termination involves both sides sending FIN packets and waiting for ACKs.

The protocol handles lost packets by retransmitting them after an initial timeout time of 2s (modified dynamically using SRTT), if the ACK is not received. This design works because the receiver sends ACKS for every received packet.

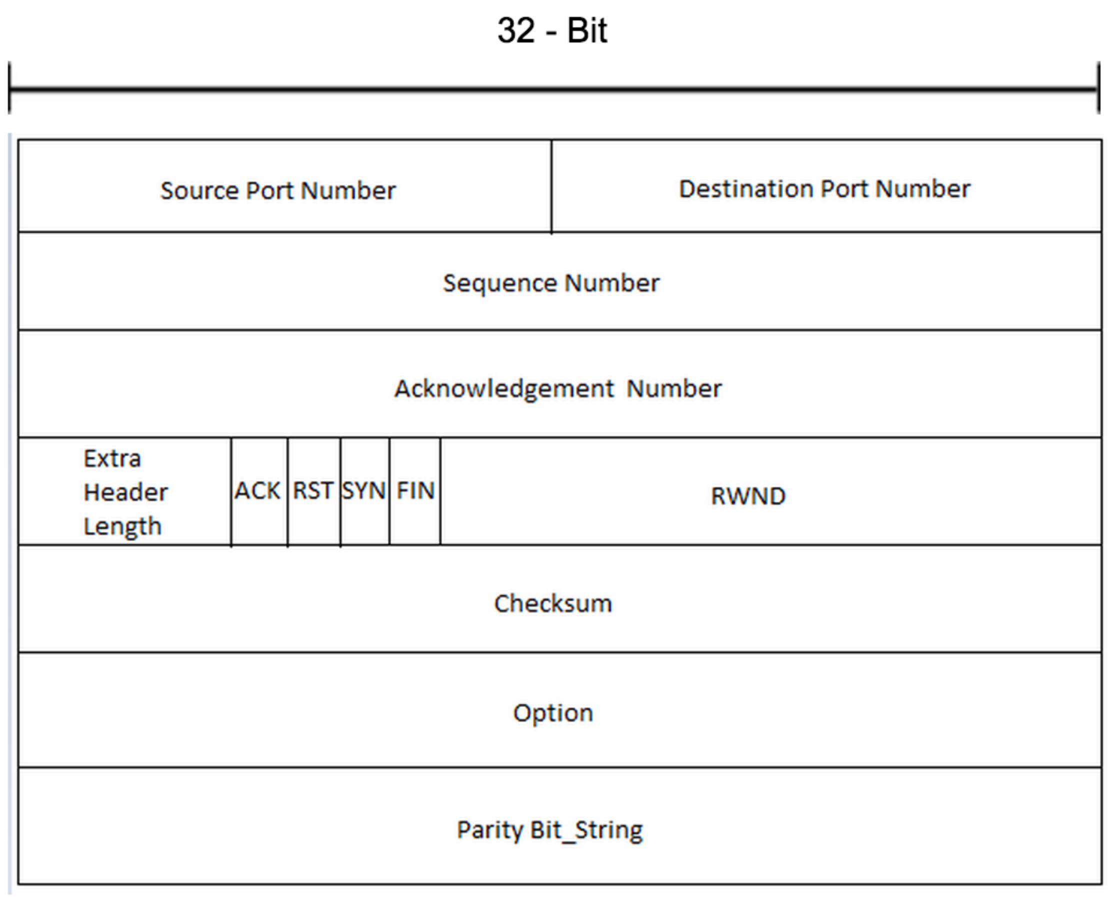
A special feature implemented in this protocol is that the user can choose how he wishes to handle corrupt packets using a command line flag. One can opt for a traditional handling of corrupt packets wherein the receiver simply drops the corrupt packet by checking the checksum. The other option is to activate Forward Error Correction(FEC) described below. This will add redundant information in the every Packet header and enable the receiver to correct bit errors without dropping the packet. Another special feature included in the protocol is the calculation of a non-trivial checksum. We are using the BSD checksum algorithm in order to calculate this non-trivial checksum. Third important function enhancement provides in-flight RSA encryption of data.

The protocol handles simultaneous bidirectional data transfer using the technique explained in the section above.

RxP Packet Flow Diagram:



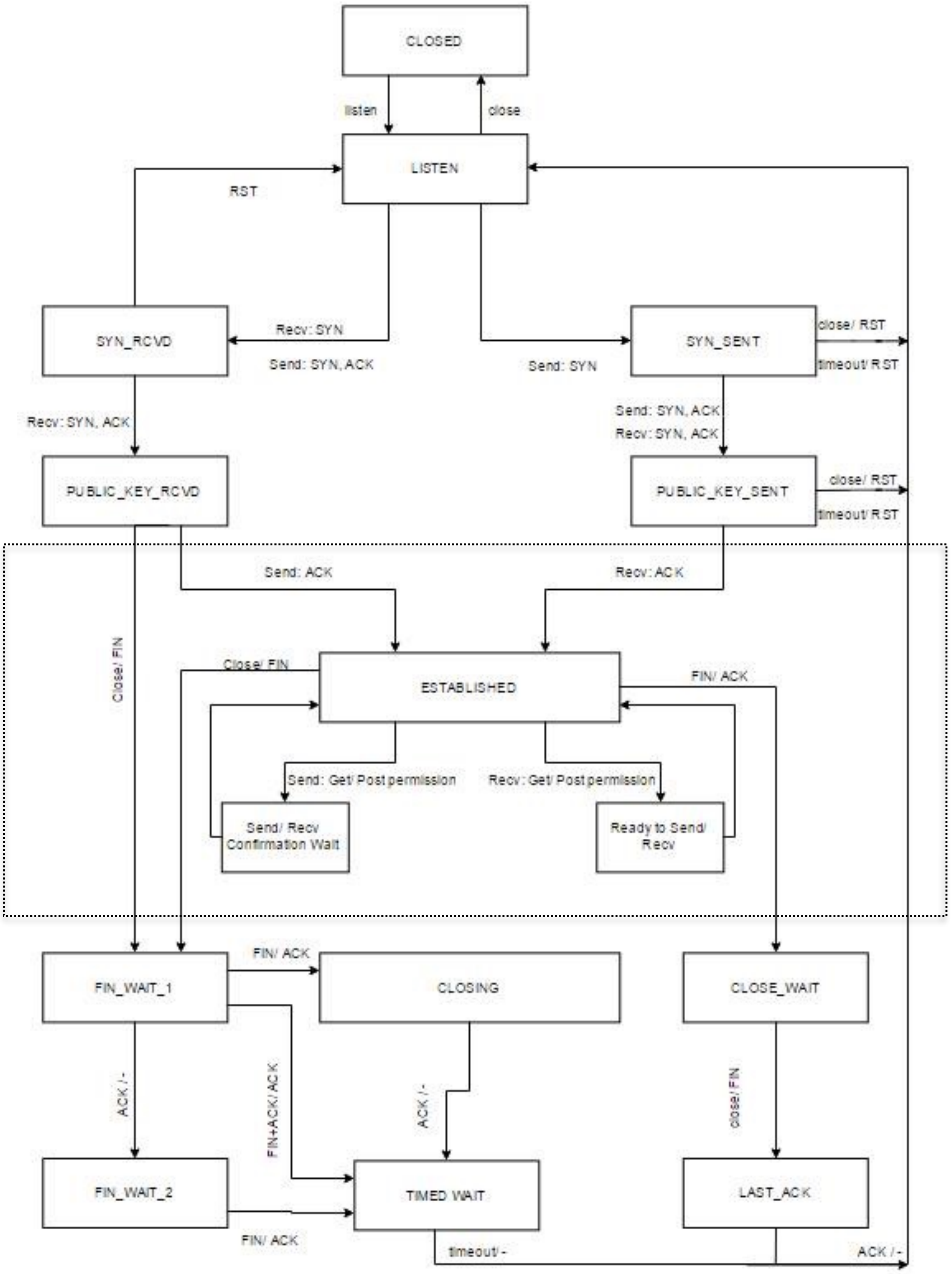
Header Structure:



The header is very similar to TCP's header with a few additions and deletions. RWND is the Receive Window Size
Deletions: The URG and PSH flags were removed as RxP does not incorporate urgency needs and these flags are used very scarcely anyway.
Additions: The checksum field was extended since there is no need for an urgent pointer anymore.
A 32 bit field for the parity matrix has been added to enable forward error correction.

Header Field descriptions:
Source Port Number: Port number of the sender.
Destination Port Number: Port number of the destination machine (receiver).
Sequence Number: Unique sequence number of a packet within a window.
Acknowledgement number: Sequence number of the next expected packet.
ACK, RST, SYN, FIN: Flags similar in function to TCP flags
RWND: Receiver's remaining buffer window size used for window based Flow-Control
Checksum: 32- bit BSD Checksum. The non-trivial checksum for data.
Parity Bit-string: Bit string used for forward error correction at the Receiver's end.

State Diagram:



API Description (functions RxP exports to the application layer):

Server:
The server implementation does not have any methods. When the serve file is started(with a port number), it simply binds to the port and listens for incoming connections, and subsequently instructions from the client.

Once a SYN request is received, a SYN ACK is sent back to the client. Then the server waits for a SYN ACK2 from the client, which contains the client's public key. The server then responds with another ACK that contains its(the server's) public key. The client then tells the server whether it is making a GET or POST request. If it is a get request, the client specifies the file it wants and the server obliges. In case of a post request, the server listens and prepares a receive buffer for the incoming file.

Client:
connect: A socket is created and bound to the clientPort and a connection request is sent to the specified IP address and the client port information is sent as provided. After the first SYN-SYN-ACK exchange, the client sends it RSA public key with SYN-ACK2 and waits for the server's ACK containing its public key. Then the client waits for its next command.

get <filename>: Using this method, the client informs the server that it is requesting <filename>. The client then gets prepared to receive and the server starts sending the file.

post <filename>: Using this method, the client informs the server that it is sending <filename>. The server then gets prepared to receive and the client starts sending the file.

disconnect: This initiates a 4 way termination handshake between the two communicating entities.

Description of files provided:
NetEmu.py - the provided net emulator file
client.py - the protocol's client side implementation
server.py - the protocol's server side implementation
FxA-client.py - FxA's client implementation
FxA-server.py - FxA's server implementation
—-NOT USED AS A SEPARATE FILE— fec.py - file containing forward error correction
—-NOT USED AS A SEPARATE FILE— crypto.py - file containing the rsa encryption code
—-NOT USED AS A SEPARATE FILE— checksum.py - implementation of the checksum
Sample.txt - contains sample command line output for a GET and a POST command
RxP Design Report - this design report

Additionally, 3 text files have been provided that were used for testing. Two of them (clientReceivedFile.txt and serverReceivedFile.txt) are generated by the program, and the test.txt file is what was being passed as input.

Instructions for Running File Transfer FxP:

Note: To run the files, you will need to have the python rss module installed on your machine. You can either do 'pip install rsa' or 'python -m pip install rsa' (if you have multiple versions of python), or download the package from https://pypi.python.org/pypi/rsa.

Server:

Command-line: python FxA-server.py [X] [A] [P]

The command-line arguments are:
X: the port number at which the FxA-server's UDP socket should bind to (odd number)
A: the IP address of NetEmu
P: the UDP port number of NetEmu

Client

Command-line: python FxA-client.py [X] [A] [P]

The command-line arguments are:

X: the port number at which the FxA-client's UDP socket should bind to (even number). Please remember that this port number should be equal to the server's port number minus 1.
A: the IP address of NetEmu
P: the UDP port number of NetEmu

Once the client is running, the following can be called by typing into the terminal:

connect - The FxA-client connects to the FxA-server (running at the same IP host).

get [F] - The FxA-client downloads file F from the server (if F exists in the same directory with the FxA-server program). The received file is stored in the same directory as the client program, under the name 'clientReceivedFile.txt'.

post [F] - The FxA-client uploads file F to the server (if F exists in the same directory with the FxA-client program). The uploaded file is stored in the same directory as the server program, under the name 'serverReceivedFile.txt'.

disconnect - The FxA-client terminates gracefully from the FxA-server.

NOTE: Vulnerability of the code: If any command is formatted in a different way, the program will throw an exception and crash. The disconnected command is implemented in such a way that if anything is typed other than disconnect, the connection is terminated anyway i.e. after the connect stage, if any incorrect command is typed, the connection will terminate and both the client and the server will shut down.

NOTE 2: In a single connection, multiple simultaneous GET/POST requests are not supported. After the completion of a GET/POST, the only command option is to disconnect.