Homework 3
Reliable Transport Protocol Description

Rohan Iyengar - riyengar6
Sara Jacks - sjacks3

CRP Protocol - A New Reliable Transit Protocol

Overview

Each connection is identified by four tuples: the IP address of the server, the IP address of the client, the port number of the client, and the port number of the server. There are three steps that each connection has establishing the connection, data transfer, and resource deallocation. To establish a connection, CRP will use a the three way handshake of to ensure both endpoints are alive and able to communicate. Next, it will send data between the two connections. Data transmission is guaranteed to be in-order, lossless, and reliable. The details of these processes are specified below this general overview. This protocol detects header errors and data errors separately by applying two distinct checksums for the header fields minus the payload and the payload; NACKs are sent to acknowledge corrupt packets. It provides window-based flow control using Selective Repeat Protocol. The final step is to close to connection and deallocate any resources used to facilitate the connection (eg. buffers, reserved memory).

Protocol Specifications

1.) Our protocol is pipelined similar to the Selective Repeat ARQ protocol. It uses a window size of 5 for both the sender and the receiver. The protocol is pipelined and will not have to revert to the state before a packet is lost. Instead, it will use this pipelined Selective Repeat ARQ protocol to only retransmit lost packets after the window is full.
2.) Our protocol will handle lost packets through the use of a timeout. If a packet has timed out, it will be resent based on the occupied window. If a packet needs to be retransmitted, the sender will know after the receive window is full at 5 packets. The sender will not have received ACK(s) based on which packets are lost so the sender will retransmit these packets before moving onto later packets.
3.) Our protocol will handle corrupted packets by using the checksum passed in the header. The checksum will be calculated as outlined in question seven and under the algorithms section of this report. If a checksum comes back different from the one in the header, the receiver will send a NACK to the sender indicating which packet needs to be resent based on corruption.
4.) Our protocol will handle duplicate packets by keeping track of the ACK numbers received from the packets in the current window. If the sender receives an ACK number that it has already received, it would not add that message to the buffer to be passed back.
5.) Our protocol will handle out-of-order packets by analyzing the sequence number in the header. The receiver will have a buffer twice the size of the window size in order to accommodate for out-of-order packets. If a packet with a later sequence number is received before the one that the receiver is expecting, the receiver will send a ACK to the sender that the packet is received, but will not process the data until the packet next in line is received.
6.) In order to allow for bi-directional data transfer, the protocol will allow the socket to be both a sender and a receiver. Each instantiated socket will keep track of the ACKs received within the current window size and the last transmitted ACK. In terms of specific changes, that means each socket will keep both a send and receive buffer to be able to transmit in both directions.
7.) Our protocol plans on using a checksum based on the data and the header. To produce the checksum with the data, we use the
8.) Extra credit goes here

APIs

CRP_Socket Object API

socket(ipVersion, packetType, protocolNumber): Constructor to create a new socket with the IP version, packet types, and protocol number (which usually defaults to zero) passed in. ipVersionand packetType are mandatory, while protocolNumber is optional.

connect(address): connects a socket to another socket at the given address (a tuple containing the IP address and port number). IP Address is a String, while port number is an integer.

send(message, flags): Sends the passed in message over the connection to the connected address. Flags can also be passed in, but they are optional. Message is a String.

close(): Closes the socket object. Afterwards, the object will not be able to perform any other socket operations. Any queued packets/data are removed from the buffer.

shutdown(): Closes the connection on that socket. This allows operations to occur but send/receive operations will fail because the connection has been closed and resources terminated.

listen(numConnections): Listen on the socket with and accept the integer maximum number of queued connections.

// Doesn't matter I think accept(): Accept a single connection from the listen queue. Returns a tuple of a socket that represents the connection and the address of that socket on the other side of the connection.

recv(bufferSize, flags): Receive the amount of data equal to the passed in buffer size. Flags are optional. It returns a packet with the data encapsulating the data sent.

bind(address): Assigns the socket object to the given address (a tuple containing the IP address and port number). This address will be used by other sockets to communicate with the now bound socket.

CRP_Controller API

createAndBindSocket(ip_address, port_number): Helper method to create and bind a socket using the CRP_Socket API parameters and returns that socket.

clientSideConnect(clientSocket, ip_address, port_number): Initiates the client side part of the three way handshake by sending a SYN packet. Also initializes the sequence number and ack number to ensure in order delivery. Terminates when the corresponding SYN+ACK response is received.
listenForConnection(serverSocket, numConnections): a method that allows a socket to listen repeatedly for incoming connections.

serverSideAccept(self, serverSocket, addr_tuple): allows the server to send back the SYNACK after SYN is received. Once the ACK is received, the sequence and ACK numbers are intiialized through packet exchange.

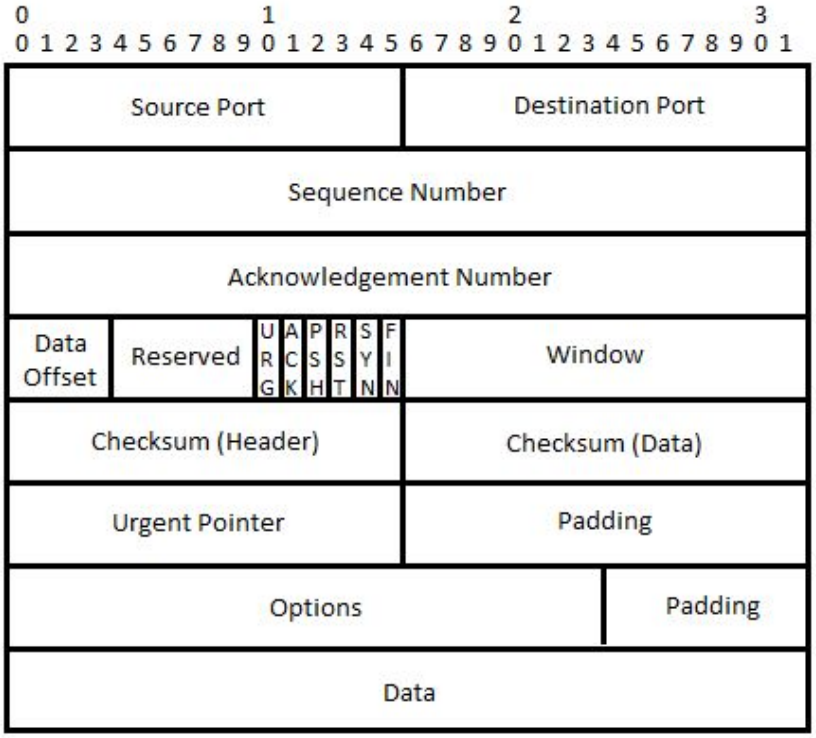
closeSocket(socket): Closes the socket passed into the method. Performs the closing handshake as described in non-trivial algorithms.

sendDataPacket(socket, size): sends a data packet using the socket given with the given size. Splits up packet based on maximum data specified for 1 packet.

recvDataPacket(): Received packets being sent. Uses sequence and ACK numbers to verify correct order of packet.

setWindowSize(socket, size): Changes the max window size of the given socket.

Header



Source Port: It is a 16-bit number that represents the source port number.

Destination Port: It is a 16-bit number that represents the destination port number.

Sequence Number: It is a 32-bit number representing where in the stream that the packet is.

Acknowledgement Number: It is a 32-bit number that contains the next expected sequence number if the ACK bit is set.

Data Offset: It is a 4-bit number stating the number of 32-bit words in the TCP header in order to indicate where the data begins.

Reserved: It is a 6-bit number that is reserved for future use and is completely zero.

Flags: It is a 6-bit field containing the control bits of the header to indicate certain information about the packet. The six 1-bit flags are an Urgent pointer field (indicating that this header field is filled), an Acknowledgement field (Stating that this packet is an acknowledgement), a push function, a reset the connection flag, a synchronize sequence number flag, and a flag to state that there is no more data from the sender.

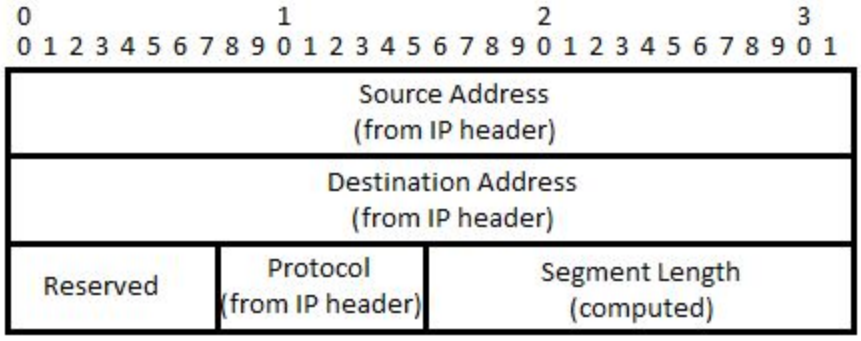
Window: It is 16-bits, and it is declaring how many packets can be accepted including the packet in the acknowledgement field.

Checksum(Header): The checksum algorithm we are using (described in algorithms section) is applied to the fields of the header minus the data, and a 16-bit checksum is generated. If this checksum does not match in future packets when the calculation is performed, that means the TCP header control fields have been corrupted.

Checksum (Data): The checksum algorithm we are using (described in algorithms section) is applied to the data and a 16-bit checksum is generated. If this checksum does not match in future packets, this means the data in the CRP header has been corrupted.

Padding: Extra 0's added to the end of the 32 bit word to ensure TCP header length is an integer multiple of 32. This allows the data offset to indicate the precise location where data begins.

<u>Pseudo-Header</u>



Source Address: This is a 32-bit IP address identifying the originator of the packet.

Destination Address: This is a 32-bit IP address identifying the intended recipient of the packet.

Reserved: This is 8-bits of zeros.

Protocol: This is a 1-bit number indicating what protocol is carried in the IP packet.

TCP Segment Length: This is computed from the length of the segment including the header and data.

Algorithms

Three Way-Handshake:

To form a connection, the server is first put in a state where it is listening for a packet. The client first sends a SYN packet (packet with syn flag set and no data). Upon receiving this packet, the client then sends a SYN-ACK packet back to the server. The server responds with a final ACK. If all of these steps are completed and the corresponding packets are sent/received on both ends, a connection is then established.

Checksum:

The checksum is calculated twice for the header and data. Since the header and data are broken down into separate sections - ie. The header has a class encapsulating all of its information and the data can be analyzed as it is a string, so this separate approach is used. A CRC-32 algorithm is used. This uses the zlib function to abstract this. To describe the algorithm briefly, it interprets the string being computed as a polynomial. Each bit is interpreted as the coefficient to a polynomial and this computes the checksum using the crc method dividing by a fixed polynomial. The unsigned integer version of the result is taken for consistency and to avoid bit errors.

Selective Repeat:

This algorithm is a pipelined process for flow control. It is a windowed algorithm using a window size of 5. Both the sender and receiver keep a window size to keep track of how many packets can be received. The software buffer implementation will be double the size, or 10 packets wide to allow lost traffic to be received in the worst case where all packets arrive out of order. The three way handshake will establish a window based on user input on both sides and set up the buffer. As packets arrive, they will be placed on the receiver's buffer and ACKs will be sent. The sender will use these ACKs to clear space on the buffer (window) as the ACKs come in order. Out of order ACKs will be acknowledged, but the appropriate buffer space will not be cleared until.the ACK comes in order. In this way, after the appropriate timeout has passed, the sender will know to resend the packets that it has not received ACKs for. This way, once this missed packet is processed by the receiver, it can process all other packets on its buffer with sequence numbers after the one it was expecting first. This way it will process the packets on its buffer before the sender removes all the packets it has been holding onto in the buffer due to the missed ACK and continue normal traffic afterwards.

Packet Exchange Mechanism:

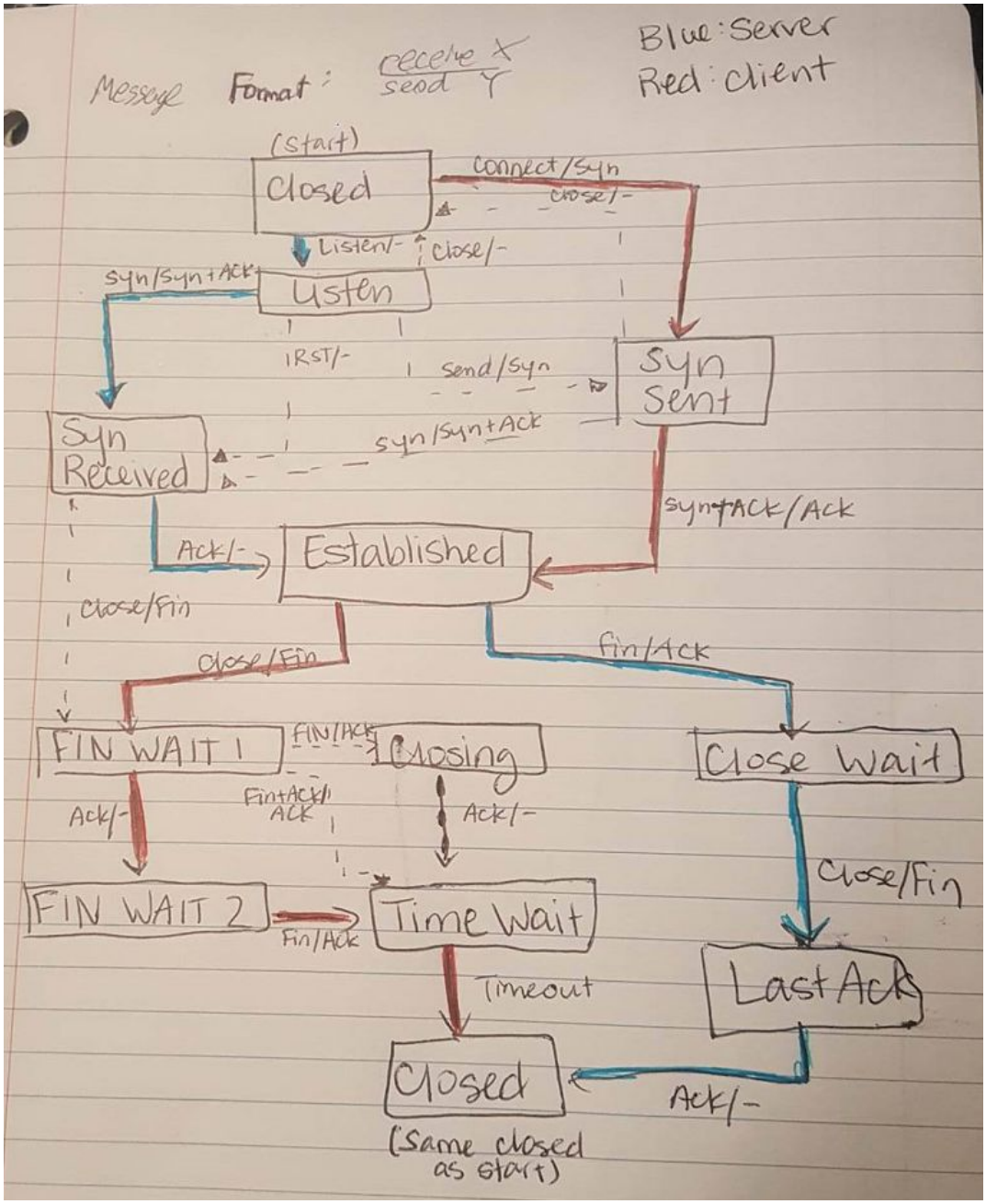
The data from a file comes out as just a raw string representing the file. The file is split based on the arbitrarily specified maximum packet size into smaller chunks and each chunk of the data is sent in a separate packet. The sequence and ack numbers of each packet are changed as each

packet is sent and a subsequent packet cannot be processed until the previous packet is received, as specified in the selective repeat algorithm.

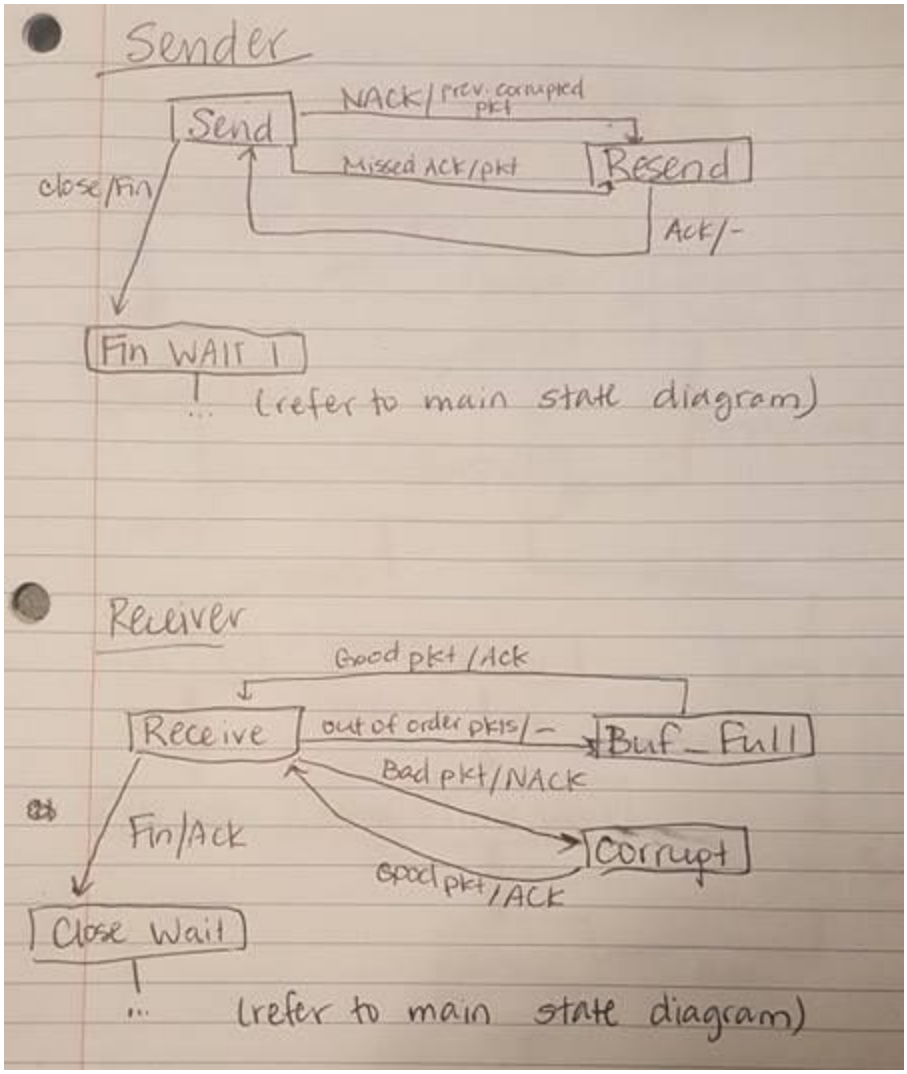
Closing Packet Flow:

This process is similar to the beginning three way handshake. The server can initiate this by sending a FIN packet. The client then replies with an ACK and the connection is closed. The client then sends a FIN packet of its own to which the server replies with an ACK. Finally the connection is closed.

State Diagram



Edge Case Handling in Established Stage:


Sender
Send
NACK/ prev. corrupted pkt
Missed Ack/pkt
Resend
close/Fin
Ack/-
Fin WAIT 1
... (refer to main state diagram)
Receiver
Good pkt / Ack
Receive
out of order pkts/-
Buf_Full
Bad pkt/NACK
Fin/Ack
Corrupt
Good pkt/ACK
Close Wait
... (refer to main state diagram)