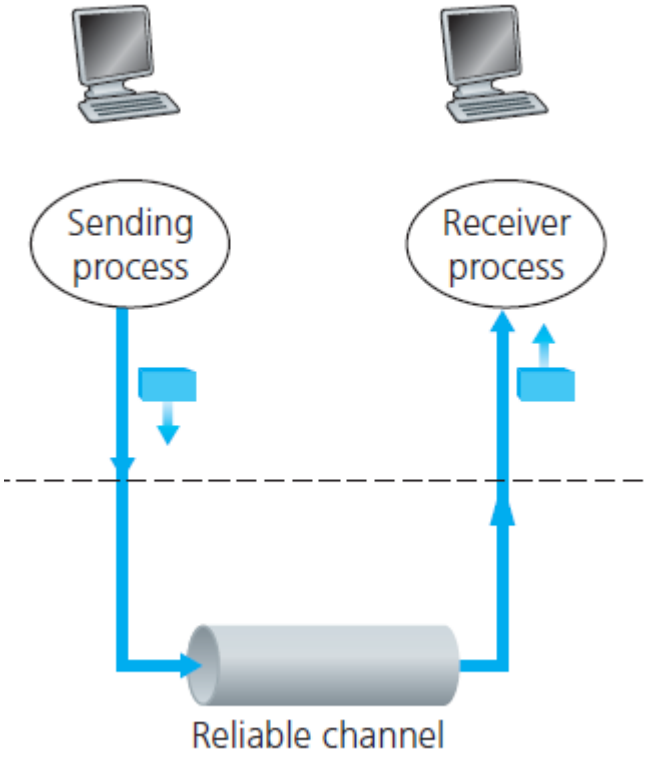
RELIABLE DATA TRANSFER PROTOCOL

Hema Bahirwani

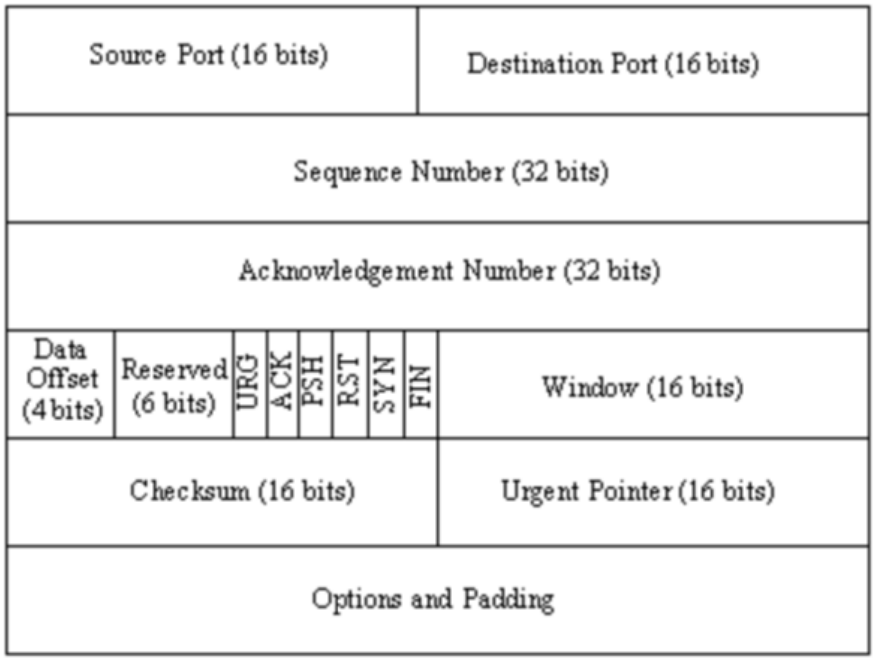
Project 2

1. INTRODUCTION:

The goal of this project is to implement a reliable data transfer protocol mimicking the basic TCP mechanism to overcome packet loss, delay, reordering, and/or corruption. The application will segment the input data, add a TCP-like header to the user data, and send the information to the server using User Datagram Protocol. The application is developed in JAVA.



2. TCP HEADER:

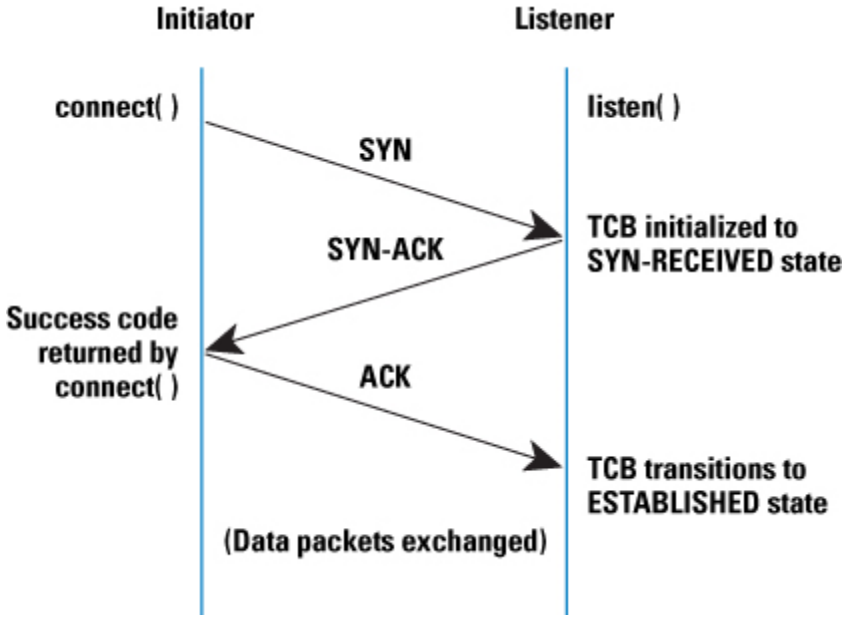


This application uses a few of the TCP header fields.

SYN: used to set up the connection
ACK: used to send an ACK
FIN: used to end the connection
Sequence Number: Used to specify the number of the packet sent
Acknowledgement Number: Used to specify the next byte to be sent
Window: to specify the receive window size
Checksum: to send across the calculated checksum, in order to avoid corruption

3. CONNECTION ESTABLISHMENT:

The client initiates the connection with the server by setting the SYN bit in the TCP header. The server then responds to this by setting the SYN bit back to the client. The client and the server randomly selects a sequence number and send across with the SYN bit. The server in addition to the two fields, also sends an ACK which is equal to the client's sequence number +1. The client then acknowledges it, also sends the payload along. This process is thus called handshake. The client resends the connection request if the server doesn't respond in a stipulated time frame. At the time of the connection, Maximum segment size is decided between the server and the client. Application data is not sent at the time while setting up the connection.



4. RELIABLE DATA TRANSFER:

TCP provides reliable data transfer that is, transferring data from one host to the other without any data loss, corruption and all the data is sent in order. This application mimics the same functionality of reliable data transfer. Once the connection is established the client sends data in chunks of the size of MSS or less, decided at the time of the setup. For each data sent, the client waits for its acknowledgement for a stipulated time. If the acknowledgement doesn't reach the client within the stipulated time period, the client resends the packet. The server receives the packets, checks for bit corruption, if corrupted, send ACK for the same packet. If the data received is not corrupted or lost, the server sends ACK for the next byte to be received. The sender sends the sequence number the receiver is expecting, and the receiver sends the acknowledgement number of the next byte it wants to receive to the sender. In case of a packet loss, server checks the sequence number it is expecting, if it is greater than what it is expecting, and with no corrupted bits, it stores in the buffer. It then sends an ACK for the packet it is expecting.

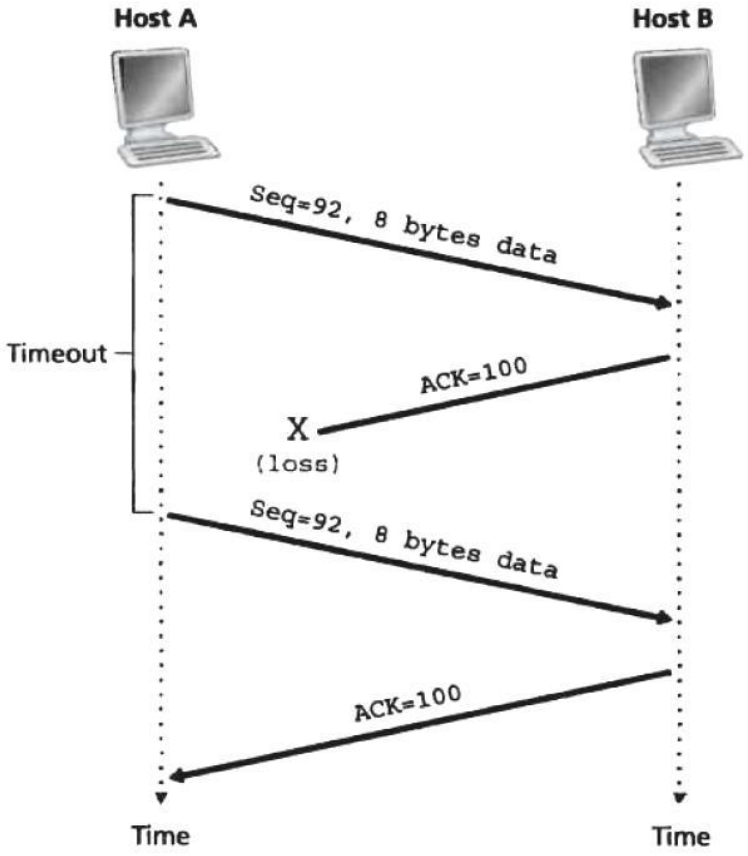
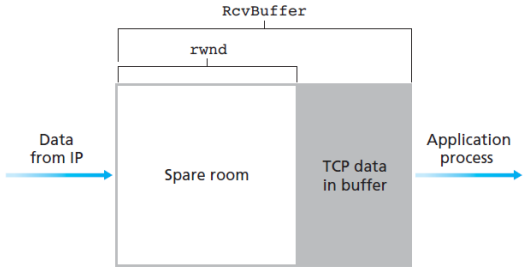
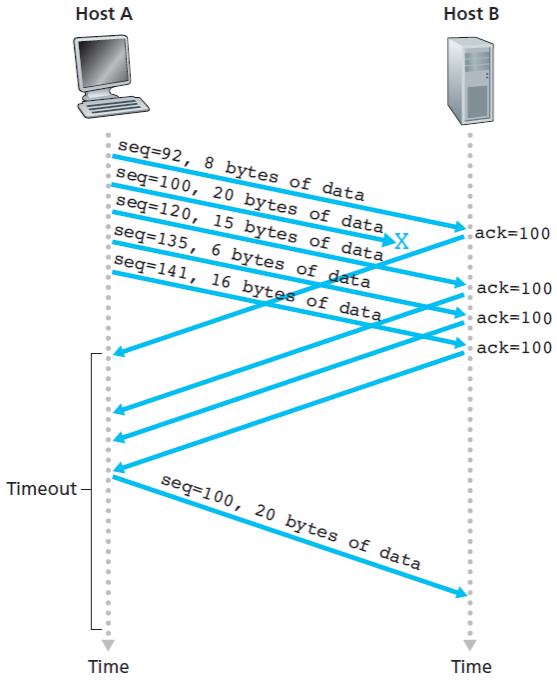


Figure 2. Retransmission due to a lost acknowledgment

5. FLOW CONTROL AND CONGESTION CONTROL

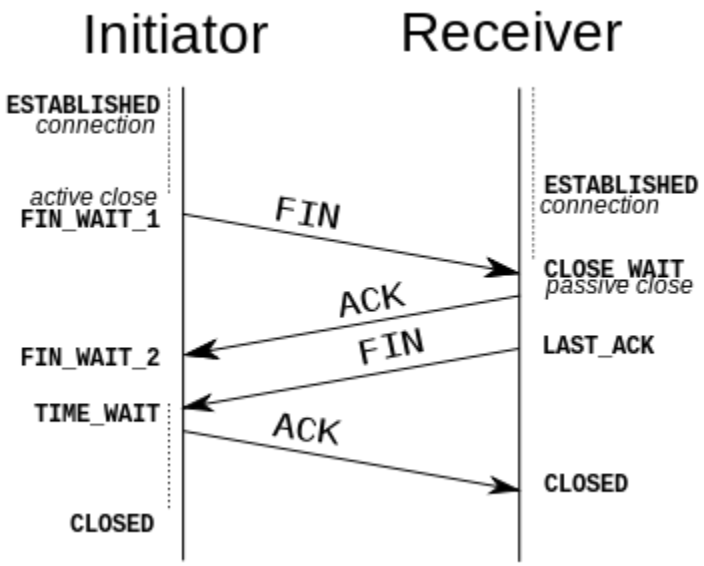
At the time of the setup, the server and the client declare buffers to receive data. If the sender sends data more than what the receiver can receive, the packets maybe lost or discarded. In order to prevent this, the receiver sends the amount of spare buffer it has that is, the amount of data it can receive. Using thing information, the sender restricts its sending rate. The sender, on the other hand, keeps a record of the number of un-acknowledged packets. Initially, the client is in the slow start phase, where the congestion window increments by 1MSS per ACK received. If the number of un-acknowledged packets exceeds the threshold, the threshold is halved and the congestion window is increased by one per RTT received. If the sender receives 3 ACKs of the same packet, then it goes in the fast retransmit phase, where it retransmits the packet back to the receiver, and increments the window by 1 MSS. If in this phase, it receives an ACK, it goes to the congestion avoidance phase. If the client doesn't receive an ACK within the stipulated time frame, the congestion window is set to 1 and the client goes in the slow start congestion phase.





6. END CONNECTION:

Once the file has been completely sent, and acknowledgement for each packet has been received, the client sets the FIN bit in the TCP header, and sends it to the server. The client enters the wait state, where it waits for the server to acknowledge and send the FIN bit. The server on receipt of this, sends an ACK back to the client. The server also sets the FIN bit, and sends to the client. Upon receipt of this, the client sends an ACK to the server, and waits for a stipulated time before ending the connection. If the server receives this packet, it shuts off the connection. But if it doesn't receive any ACK from the client, it sends the FIN back to the client. The client waits for a stipulated time to receive any packet from the server beyond which it just ends the connection.



7. CLIENT:

The client accepts various fields such as port number, server address to connect to, file path, and display style. While setting up the client, mandatory fields such as port number, server address, file path, are validated, without which there would be an exception. If the user mentions the displaying format as *QUIET*, using the command –q or –quiet, only minimal output is displayed, mainly the MD5 of the file. At the time of the setup, the client declares the congestion window (cwd) that is the number of unacknowledged packets it can allow. The client reads chunks of data of the size of MSS, calculates the checksum, attaches the checksum and the sequence number to the payload, send it to the client. The client sends *Min(cwd,rcwd)* bytes to the server, so as to avoid congestion and loss. The client then waits for an acknowledgement from the server. In this packet, the server specifies the next sequence it is expecting and the size of the receive window. Based on the congestion status, the client increments the congestion window.

8. SERVER:

The server is set up specifying the port number, which is a mandate. While setting up the server the user can mention the displaying format as *QUIET*, using the command –q or –quiet, only minimal output is displayed, mainly the MD5 of the file. At the time of the setup, the server declares a receive window that is, the number of packets it can receive at a time. It waits for the data from the client, reads the checksum and the sequence number from the header. It calculates the checksum of the payload received, and if it matches with the received checksum, also, if the sequence number matches the expected sequence number, it sends an ACK to the client, stating the next sequence and the size of the receive window. If none of the above satisfies, it sends an ACK stating the sequence number it is expecting. If the checksum is correct, but the packet is out of order, it then stores in the buffer.

9. IMPLEMENTATION ILLUSTRATION:

SERVER:

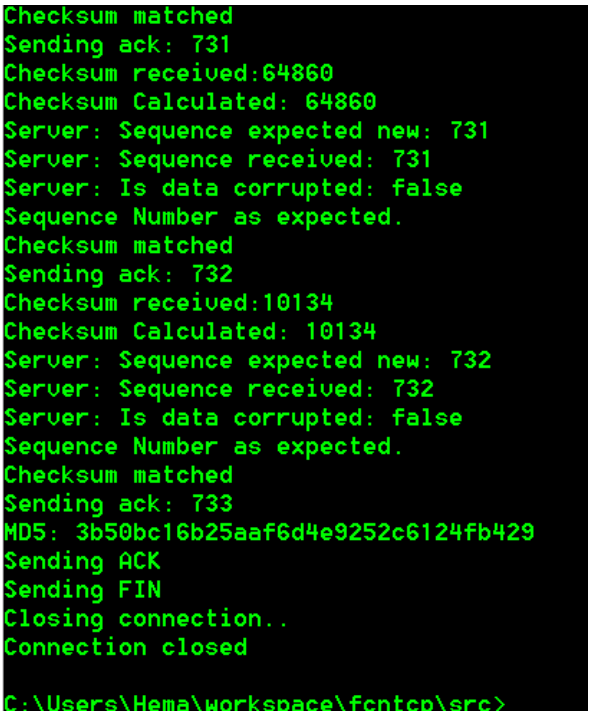
```
C:\Users\Hema\workspace\fcntcp\src>java fcntcp -s 4000 -q
Sever!
Server is listening on port: 4000
Connection Established!
MD5: 3b50bc16b25aaf6d4e9252c6124fb429
Sending ACK
Sending FIN
Closing connection..
Connection closed
```

CLIENT:

```
C:\Users\Hema\workspace\fcntcp\src>java fcntcp -c 4000 127.0.0.5 -f C:\Users\Hem
a\workspace\fcntcp\csci651_proj2_1M.bin -q
my file name C:\Users\Hema\workspace\fcntcp\csci651_proj2_1M.bin
Client!
Connection Established
MD5: 3b50bc16b25aaf6d4e9252c6124fb429
Received ACK
Received FIN
Waited for 1000
Closing Connection
Connection closed.
```

WITH FULL DISPLAY FORMAT:

SERVER:



CLIENT: