Computer Networks - II

Programming assignment - III

Saurabh (CS14BTECH11031)

Stop and Wait Protocol (SWP)

Server-side implementation :

Socket creation and host/port designation :

```
try :
    socket_id = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except Exception as e :
    print "Socket creation failed, probably already in use."
    exit(0)

filename, seq_num, ack_num, address = sys.argv[1], 0, None, (socket.gethostbyname("0.0.0.0"), 5000)
```

The socket is created using the **socket.socket([family[, type[, protocol]]])** call, and the host is specified as 0.0.0.0 which means that the server can listen on all IP addresses on the local machine, not only the localhost which is 127.0.0.1. In simple words, this means that the server can listen on all the configured network interfaces.

The **socket.socket([family[, type[, protocol]]])** opens a socket of the given type. The family argument specifies the address family which defaults to AF_INET in our case. The type argument specifies whether this is a stream (**SOCK_DGRAM** in our case since we are using UDP) or TCP (SOCK_STREAM) socket. Socket.socket is basically a method of the **_socketobject(__builtin__.object)** which has other class methods such as *send()*, *sendall()*, *recv()*, *accept()*, *bind(), close()*, *shutdown()*, etc.

After this, the server waits for a dummy request from the client in order to receive the address of the client used in **socket.sendto()** method.

```
while True :
    try :
        test_recv, address = socket_id.recvfrom(1024)
        print "Client requested the server to download the file !"
        break
    except socket.timeout as e :
        print "Waiting for the client to send a request ..."
        continue
```

After this, the alternating-bit protocol starts now. The below is the code that does this :

```
while True :
    start = time.time()
    socket_id.sendto(packet, address)

    while True :
        try :
            recv_packet, address = socket_id.recvfrom(1024)
            end = time.time()
            SampleRTT = end - start
            EstRTT = 0.875*EstRTT + 0.125*SampleRTT
            # set the timeout here to 3*EstRTT
            socket_id.settimeout(3*EstRTT)
            ack_num = get_ack_no(recv_packet)
            if seq_num == ack_num :
                print "Packet with sequence number", seq_num, "delivered !"
                break
            else :
                print "ACK not equal to SEQ or data corrupted !"
        except socket.timeout :
            print "Timeout occured ! Resending packet ..."
            start = time.time()
            socket_id.sendto(packet, address)

    data = fp.read(1000)
    if data == '' :
        packet = "END"
        socket_id.sendto(packet, address)
        print "File transfer complete. Server now shutting down."
        break
    seq_num = (seq_num + 1) % 2
    packet = get_packet(seq_num, data, "not_first")
```

The server sends the filename and size of the file in the first packet with SEQ = 0, and waits for an ACK of 0 from the client. The inner **while** loop tries to keep receiving from the socket. If a timeout occurs, the start time is updated and the previous packet is sent again. If no timeout occurs, and a successful receive is made by the server, then RTT is calculated, and the exponential average formula is used to calculate the EstimatedRTT with alpha = 0.125. If the SEQ is same as ACK, success message is printed along a **break** from the while loop, else the loop continues. Now after this, further portion of the file to be send is read and stored in the packet, and the SEQ is changed to the alternative bit (changes to 0 if was 1 and vice versa).

At the end, if no further data from the file can be received, the server sends a special packet to the client indicating that the file transfer is complete and the server then stops its execution, after closing the socket and the file.

Client-side implementation :

The socket setup and assigning of address IP along with the port is the same in the client side as the server. The following is the crux of the client code :

```
while True :
    try :
        data, address = socket_id.recvfrom(4096)
        if data == "END" :
            break
        recv_packet, content = struct.unpack('HHL', data[:16]), data[16:]
        seq_num, check_sum, length = recv_packet[0], recv_packet[1], recv_packet[2]
        if corrupt(data) or ack_num != seq_num :
            send_seq_num = 1 if ack_num is 0 else 0
            socket_id.sendto("ACK" + str(send_seq_num), address)
            print "ACK not equal to SEQ or data corrupted !"
        elif data[-1] == "|" and data[-2] == "|" :
            file_name = "rcvd_" + data[16:len(data) - 2]
            try :
                os.remove(file_name)
            except OSError :
                pass
            fp = open(file_name, "wb")
            file_size = length
            print "File '" + file_name + "' with file size", file_size, "bytes opened for downloading !"
            ack_num = (ack_num + 1) % 2
            print "Sending ACK numbered", ack_num, "to the server ..."
            socket_id.sendto("ACK" + str(seq_num), address)
        elif not corrupt(data) and ack_num == seq_num :
            fp.write(content)
            ack_num = (ack_num + 1) % 2
            print "Sending ACK numbered", ack_num, "to the server ..."
            socket_id.sendto("ACK" + str(seq_num), address)
    except Exception as e :
        print "Exception :", e
        pass
```

The client starts receiving packets from the server. The first check is done for the special packet indicating that the server has terminated and the client terminates accordingly. Then, general packet parsing is done to extract headers like checksum, length of the payload, filename/content, etc. and once the information is obtained, appropriate checks are performed.

If the data is corrupt where we just compare the checksum sent in the packet with the one we newly form in order to resemble a packet which was originally sent from the server or SEQ is not equal ACK, error message is sent a packet is sent to the server again, requesting for the same packet to be sent again. If the data is not corrupt, and the packet sent is the first one (I append the string "||" in the payload indicating that the packet is the first one), then we parse the packet accordingly and obtain the file name from the packet, give the same name to it, modified a bit by adding "rcvd_" to the beginning of the name of the file to be downloaded and start writing data to this file from next time onwards. In the same case, the ACK is flipped and sent to the server, thus requesting for the next packet in line. In the last case, when neither the data is corrupted and the ACK is equal to the SEQ sent from the server, the data obtained is written to the file opened, the ACK is flipped and again as in the previous case, the client sends this packet, thus requesting the server to again send the next packet in line.

Once the client receives the special end signifying packet, the client terminates the infinite **while** loop, closes the socket and the file and ends its execution.

Now we see some of the common methods used in both all the server/client programs in both the protocols.

Packet creation :

```
def get_packet(seq_num, payload, put) :
    header = struct.pack('HHL', seq_num, 0, 0)
    my_checksum = checksum(header + payload)
    if put == "first" :
        length = os.path.getsize(filename)
    else :
        length = len(payload)
    header = struct.pack('HHL', seq_num, my_checksum, length)
    return header + payload
```

We create a packet as a **struct** pack in python, where we encode the header as the struct members, and append a payload, which is basically the content we are sending in the string format, to this struct. This method returns the UDP header followed by the payload containing the content. Note that length of payload is appended in all cases except the case where we

send the packet for the first time to the client, where it stores information about the file being sent.

Checksum calculation :

```
def carry_around_add(a, b) :
    c = a + b
    return (c & 0xffff) + (c >> 16)

def checksum(msg) :
    s = 0
    if len(msg) % 2 != 0 :
         msg += "\x00"
    for i in range(0, len(msg), 2) :
        w = ord(msg[i]) + (ord(msg[i + 1]) << 8)
        s = carry_around_add(s, w)
    return ~s & 0xffff
```

We just keep adding the corresponding value for the byte encountered in the file stream, and finally return the BITWISE AND of the negation of the byte sum with -1 (0xffff in hexadecimal).

Packet-corruption check :

```
def corrupt(packet) :
    if packet[-1] == '|' and packet[-2] == '|' :
        return False
    seq_num, check_sum, length = struct.unpack('HHL', packet[:16])
    header_new = struct.pack('HHL', seq_num, 0, 0)
    return check_sum != checksum(header_new + packet[16:])
```

The above method extracts the checksum from the sent packet, and compares it with the checksum of a duplicate packet created at the client side to check whether the checksum of the new packet calculated is equal to the sent value from the server side.

Sliding Window Protocol (Go-Back-N) - GBN

Server-side implementation :

Socket creation, socket binding and address/port setup is basically same as in the case of a Stop-and-Wait server. There are basically two steps in the server case : First, sending all the packets in the current window until the window is not empty, and, Second, waiting for an ACK from the client, and in any case of corrupt packet/unordered packet received by the client, the server sending the window of packets starting from the last unacked packet to the current packet.

```
enter = 0
while nextseqnum < base + N and nextseqnum < num_of_packets :
    enter += 1
    data = filename + "||" if base is 0 and nextseqnum is 0 else fp.read(1000)
    packet = get_packet(nextseqnum, data, "first") if base is 0 and nextseqnum is 0 else get_packet(nextseqnum, data, "not_first")
    sndpkt.append(packet)
    print "Sending packet with sequence number", nextseqnum, "and size", len(data), "!"
    while True :
        try :
            start = time.time() if enter is 1 else start
            socket_id.sendto(packet, address)
            break
        except socket.timeout as e :
            continue
    nextseqnum += 1
```

In the above code snippet, the **while** loop runs until the window is empty, that is, our counter reaches the end of the window, and sends all the packets in the current window. Note, in the first packet transmission, we just send the name followed by the special string “||”, indicating that this packet is the first one, and contains the information about the file like the filename and Filesize. Once the packet is formulated, its appended in the **sndpkt** list, used for later re-transmissions in case a timeout occurs. Here, **nextseqnum** is the last unacked and **base** being the packet next to the one which was sent last.

Once the server is done sending all the packets (and incrementing the value of last unacked packet sequence number), it switches to the receiving mode and waits for an ACK from the client. Each time a packet is sent with a sequence number, SEQ, it waits for an ACK = SEQ from the client, and then, in case of a successful ACK received from the server, both ACK and SEQ increment by 1. Note that SEQ already gets incremented by 1 when it gets appended in the header of the packet corresponding to that sequence number, SEQ. At the server side, the value of the variable **base**, which represents the beginning of the sending window becomes equal to [ACK (received by the client) + 1], which is basically the last sent packet.

This happens in the below code snippet :

```
try :
    recv_packet, address = socket_id.recvfrom(1024)
    end = time.time()
    SampleRTT = end - start
    EstRTT = 0.875*EstRTT + 0.125*SampleRTT
    # set the timeout here to 3*EstRTT
    socket_id.settimeout(3*EstRTT)
    ack_num = get_ack_no(recv_packet)
    if ack_num + 1 == num_of_packets :
        print "File transfer complete. Server now shutting down."
        break
    if ack_num == -1 :
        # packet is corrupted
        print "Packet received is corrupted !"
    else :
        print "ACK numbered", ack_num, "received from client !"
        base = ack_num + 1
except socket.timeout :
    print "Timeout occured ! Resending packets' number ranging from", base, "to", nextseqnum - 1
    for i in range(base, nextseqnum) :
        print "Sending packet with sequence number", i, "and size", len(sndpkt[i]), "!"
        while True :
            try :
                #start = time.time() if i == base else start
                socket_id.sendto(sndpkt[i], address)
                break
            except socket.timeout as e :
                continue
```

Note that the timeout calculation using the estimated RTT is the same as was in the case of the Stop-and-Wait protocol. Extra cases of corrupted ACK received from the client are handled as well.

Note that in case of timeout, we send all the packets again, currently in the sending window, which happens under the **except socket.timeout** case.

The server exits from the **while** loop once the ACK number received from the client increments to a point that it becomes equal to the number of packets, which were decided initially to be sent to the client. Also, at this time, the server sends an info in the packet indicating that the packet being sent is the last one (function doing this given below). Once the server exits from the while loop, it closes the socket and the file and ends its execution.

```
def get_packet(seq_num, payload, put) :
    terminate = True if seq_num == num_of_packets - 1 else False
    header = struct.pack('HHHL', seq_num, terminate, 0, 0)
    my_checksum = checksum(header + payload)
    if put == "first" :
        length = file_size
    else :
        length = len(payload)
    header = struct.pack('HHHL', seq_num, terminate, my_checksum, length)
    return header + payload
```

The value of the variable **terminate** is always False but true in the case when SEQ becomes equal to the (number of packets - 1), a value of *terminate* = *True* is sent to the client, which the client receives while parsing the packet and then ends its execution as well.

Client-side implementation :

The socket setup and assigning of address IP along with the port is the same in the client side as the server. The following is the crux of the client code :

```
while not end :
    try :
        data, address = socket_id.recvfrom(4096)
        recv_packet, content = struct.unpack('HHHL', data[:16]), data[16:]
        seq_num, terminate, check_sum, length = recv_packet[0], recv_packet[1], recv_packet[2], recv_packet[3]
        if seq_num == expectedseqnum and not corrupt(data) :
            expectedseqnum += 1
            if data[-1] == "|" and data[-2] == "|" :
                file_name = "rcvd_" + data[16:len(data) - 2]
                try :
                    os.remove(file_name)
                except OSError :
                    pass
                fp = open(file_name, "wb")
                file_size = length
                print "File '" + file_name + "' with file size", file_size, "bytes opened for downloading !"
            else :
                fp.write(content)
                print "Packet with sequence number", seq_num, "received !"
            end = terminate
        else :
            print "Packet with sequence number", seq_num , "discarded (corrupted or not in order) !"
        print "Sending ACK numbered", expectedseqnum - 1, "to the server ..."
        socket_id.sendto("ACK" + str(expectedseqnum - 1), address)
    except Exception as e :
        print "Exception :", e
        pass
```

The **while** loop runs until the variable **end** becomes **True**. At the end of a successful packet receipt, **end** becomes equal to the value of **terminate** sent from the server. Hence, when the server sends *terminate* = *True* to the client, indicating that the server has finished sending all the packets, the value of **end** becomes **True** as well, hence creating a **while not True** condition, and hence, the client then exits the while loop.

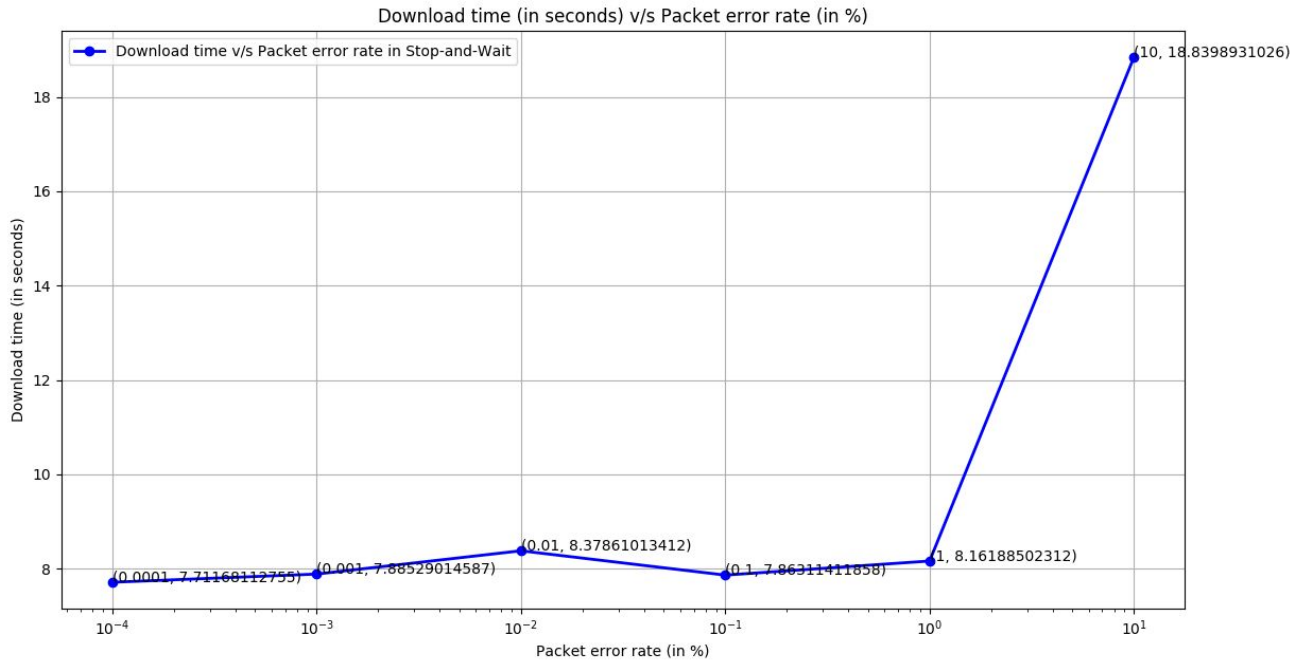
The client receives the packet, parses it, storing the filename/content being sent in the packet in appropriate variables, and getting the value of SEQ as well. The client checks whether its expected sequence number is equal to the SEQ sent from the server, and if yes, the client increments its expected sequence number and sends this sequence number - 1 in the ACK to the server, thus, requesting for the next series of packets. In all other timeout and corrupt/unordered packet cases, the client sends the unchanged expected sequence number in the ACK to the server.

Once the client exits from the while loop as explained above, the client closes the socket and the file and ends its execution.

Studies performed and inferences made :

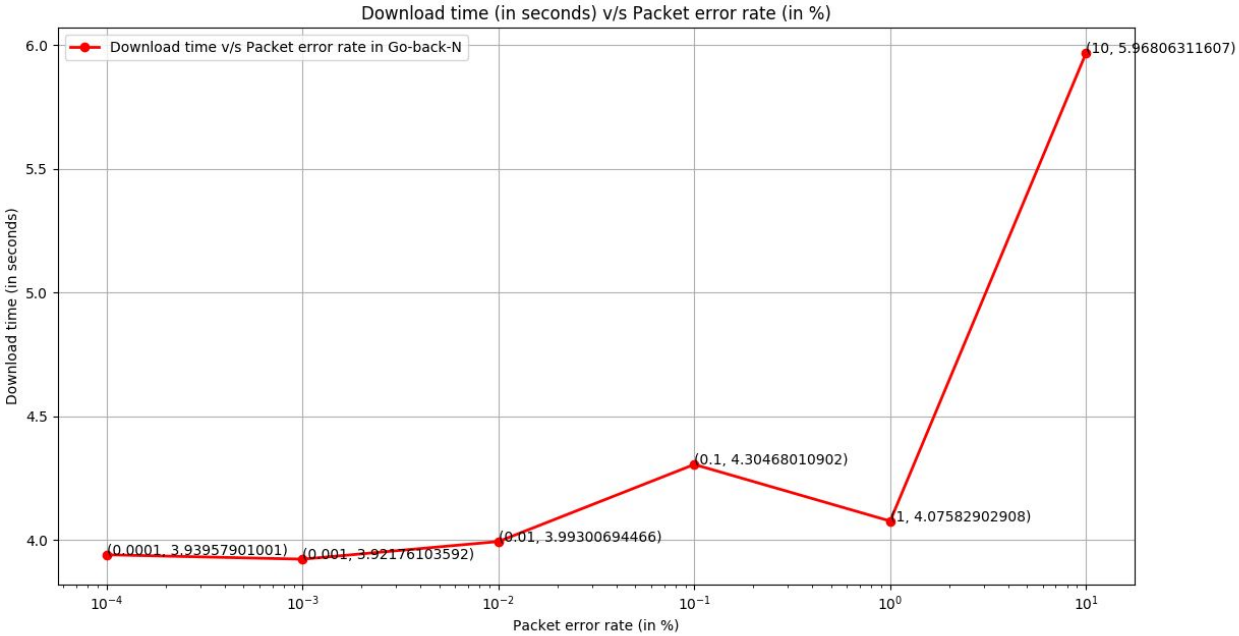
(a) Varying the packet corruption/error rate from 10^-6 to 10^-6 and plotting the time to download a fixed size file for both SWP and GBN-5, GBN-10, GBN-15, GBN-20 :

SWP :

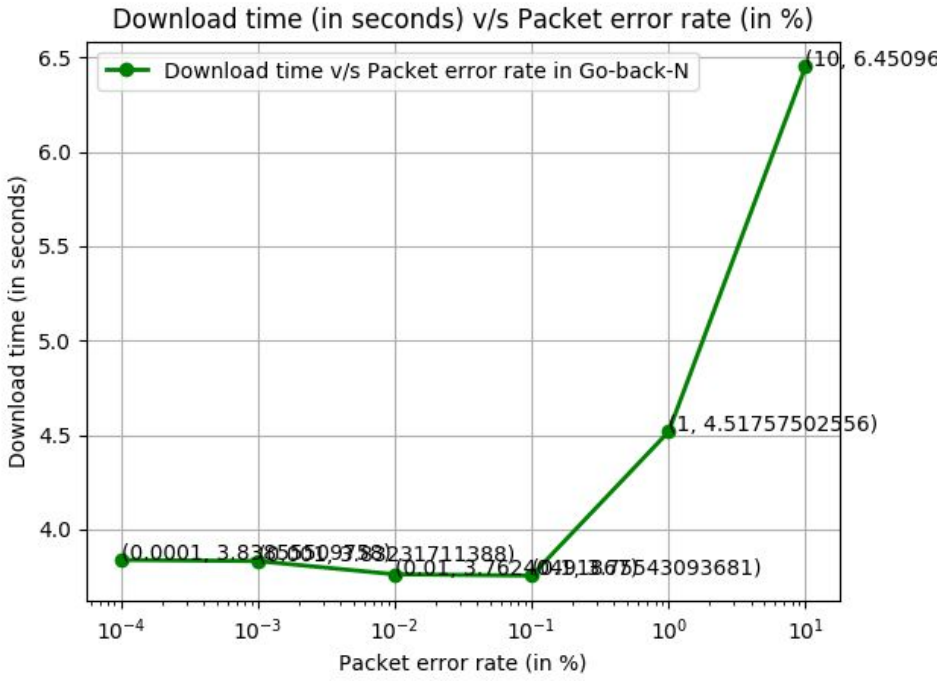


From the above graph, we observe that the download time in a Stop-and-Wait protocol generally increases, and for a packet error rate of 10^-1, the download time increases substantially. We download a file of size 10 MB in both the SWP and the below GBN protocol.

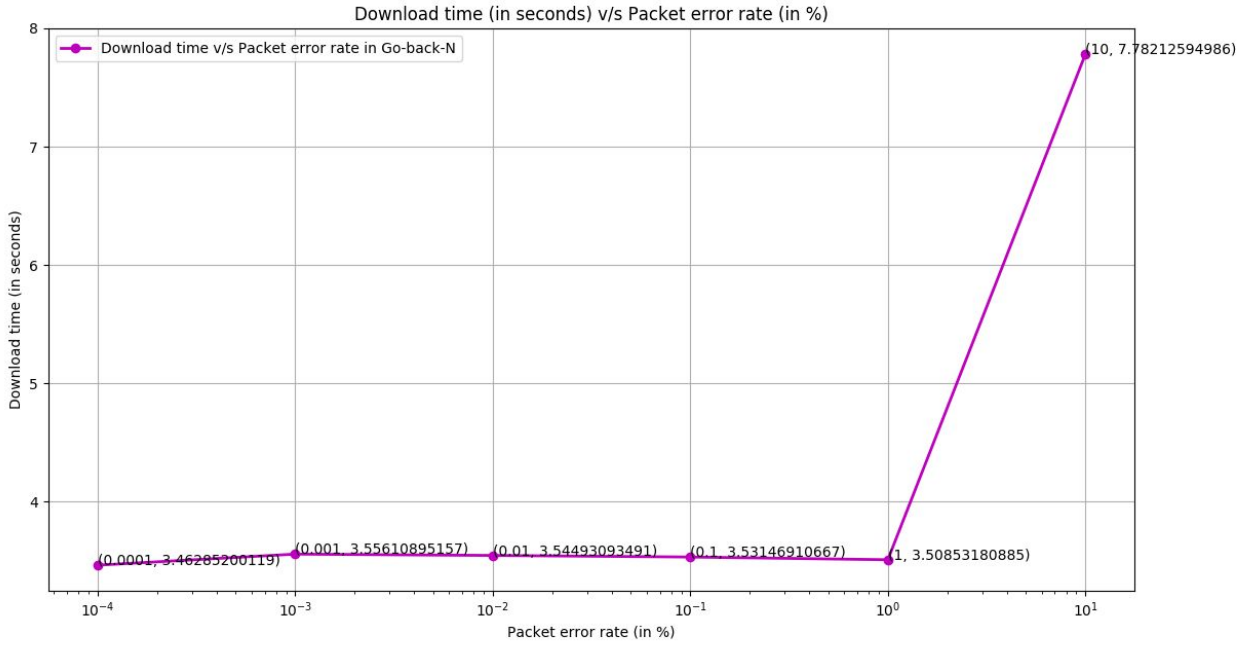
GBN-5 :



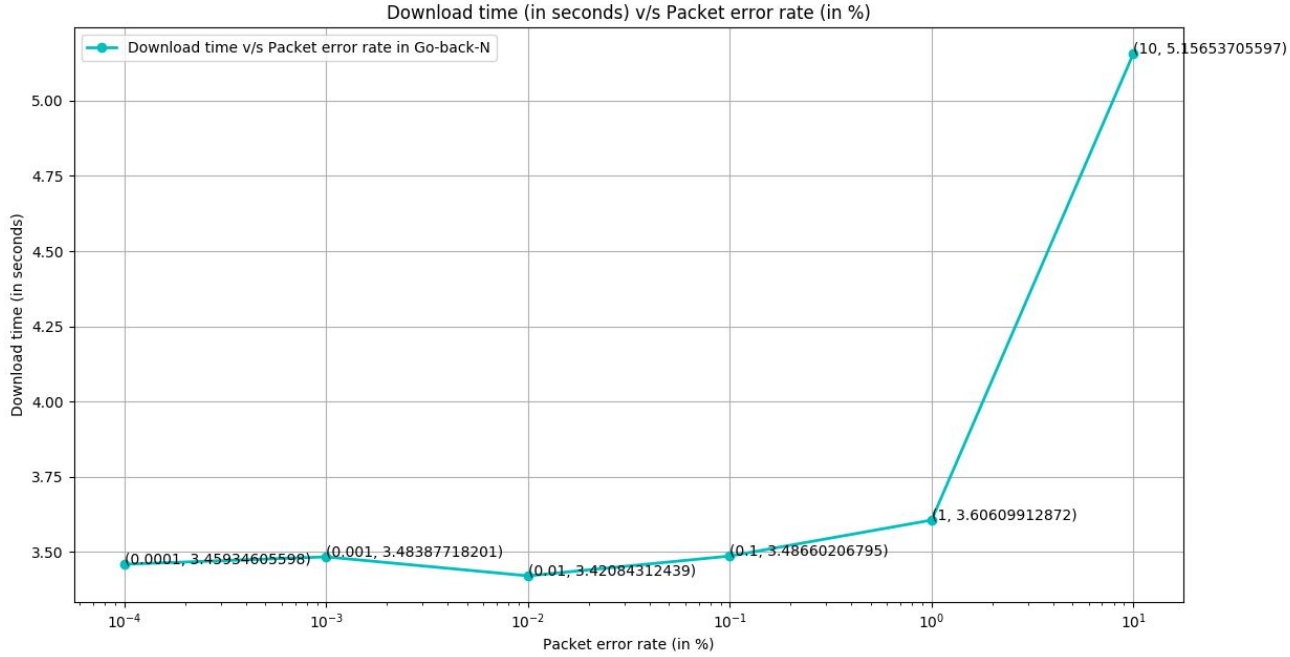
GBN-10 :



GBN-15 :



GBN-20 :

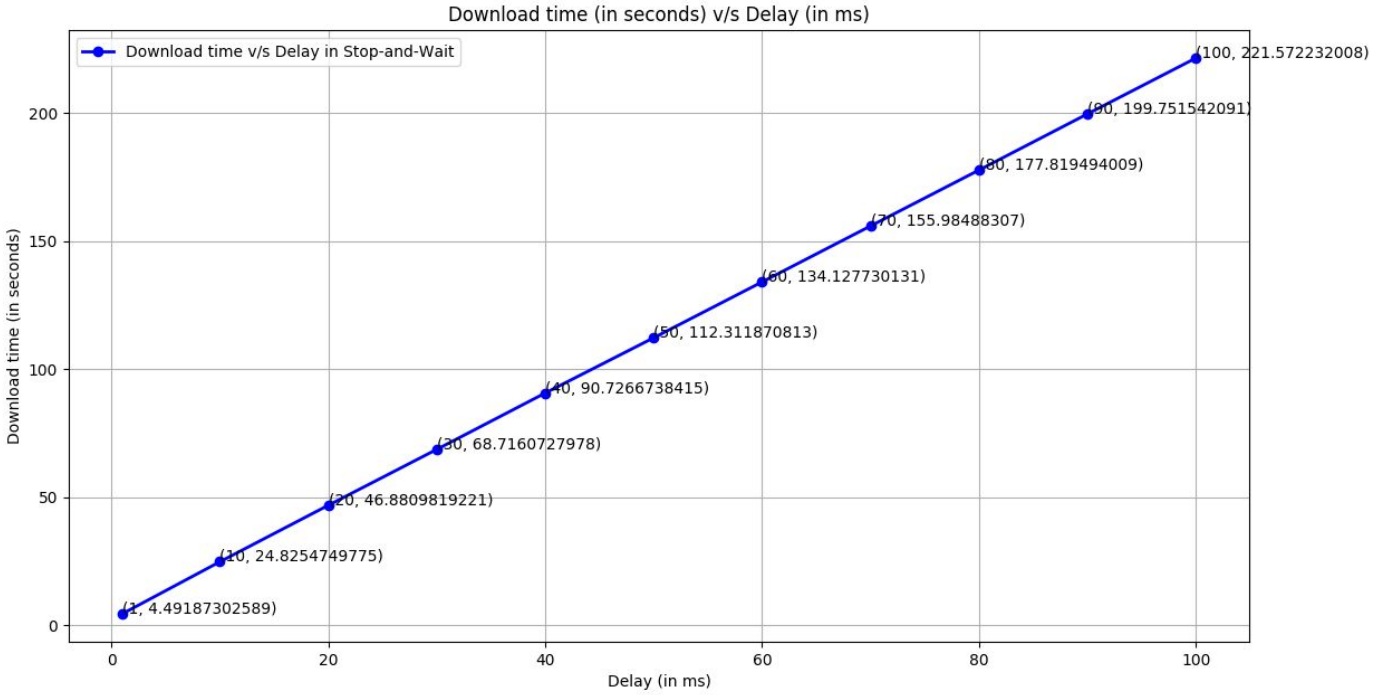


Note that the pattern in all the Go-back-N downloads is the same, that is, the download time keeps increasing generally, and for the higher packet error rate, the download time increases

substantially in the all the cases. Another important observation is that Go-back-N is **faster** (and efficient as well) than the Stop-and-Wait protocol and also, among all the GBN downloads, the fastest overall downloads for the varying packet error rates is observed when N = 5.

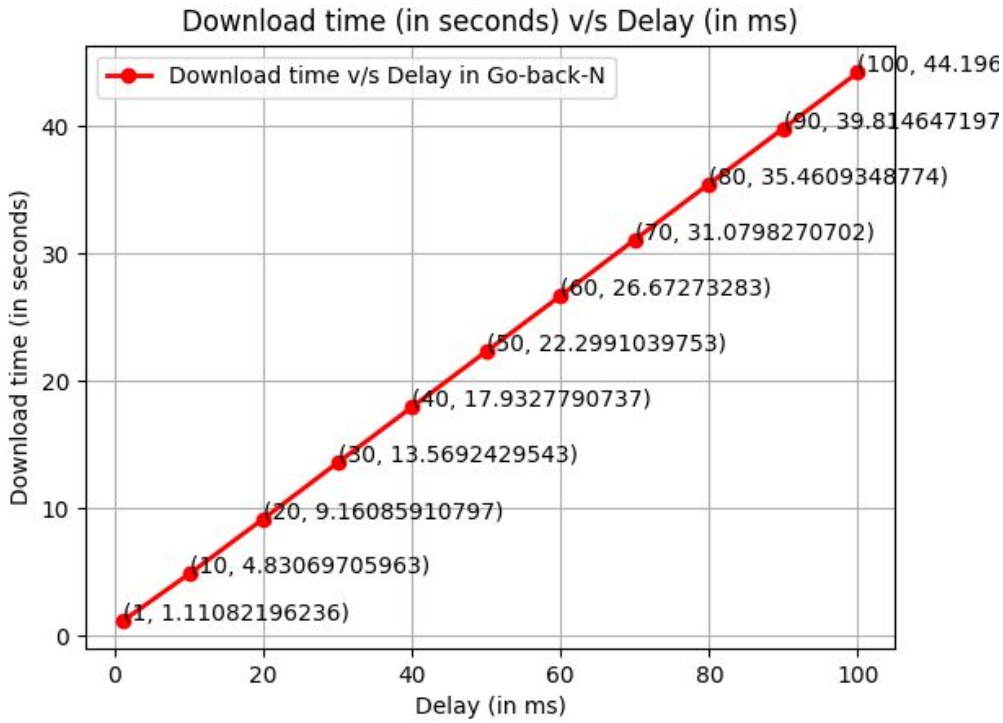
(b) Varying the packet delay from 1 msec to 100 msec (at step size of 10 msec), for a fixed packet loss rate of 10^-2 , and plot the time to download a fixed size file for both SWP and GBN-5, GBN-10, GBN-15, GBN-20 :

SWP :



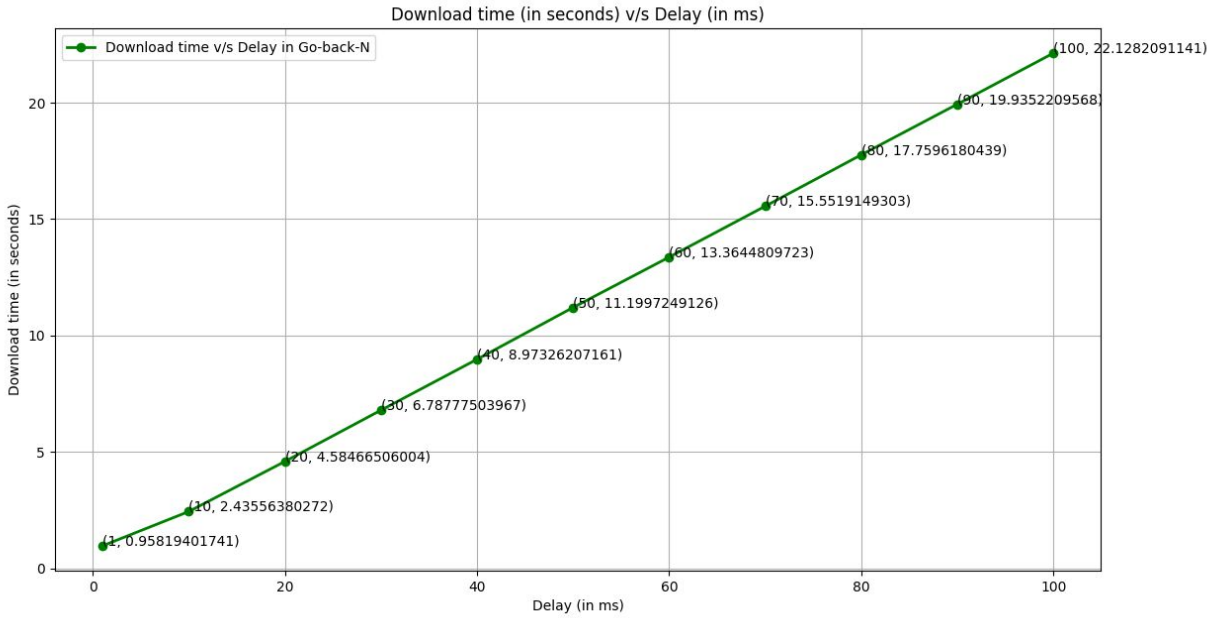
Note that the download time increases linearly with the packet delay, and reaches a maximum of 221 seconds for a file download of 3 MB.

GBN-5 :



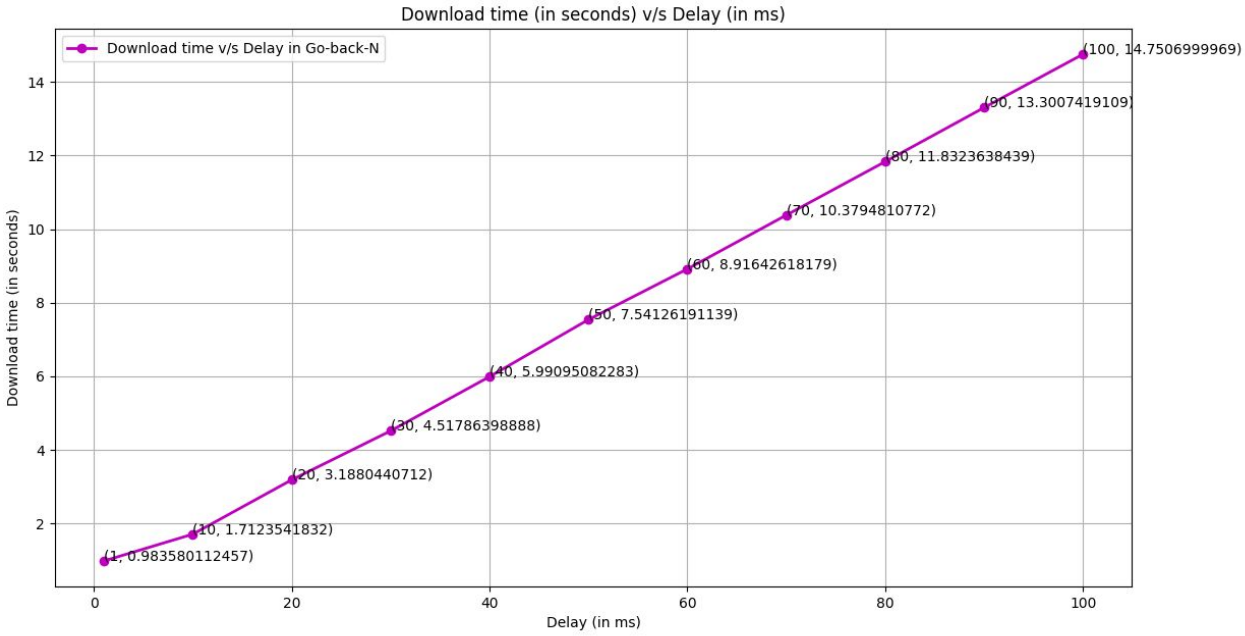
One thing we observe along the linearity is that the download is almost 5 times faster in this case as compared to the previous Stop-and-Wait protocol case.

GBN-10 :

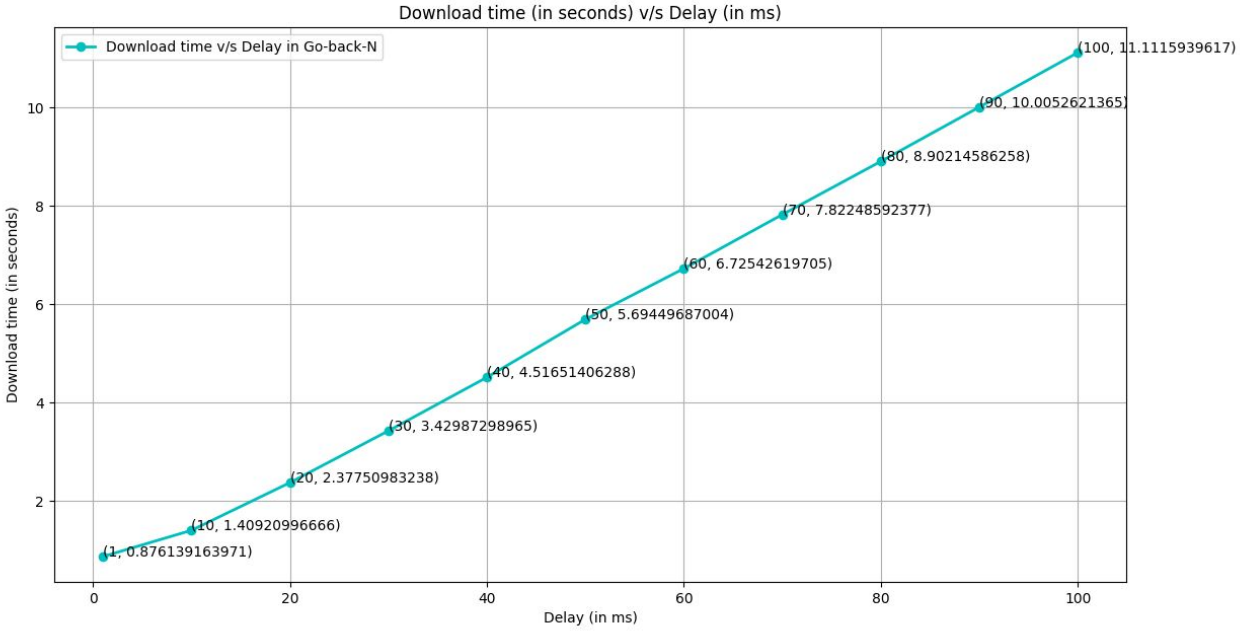


Again, now in comparison between GBN-5 and GBN-10, the download speed in the second one is twice as much as in the first case.

GBN-15 :



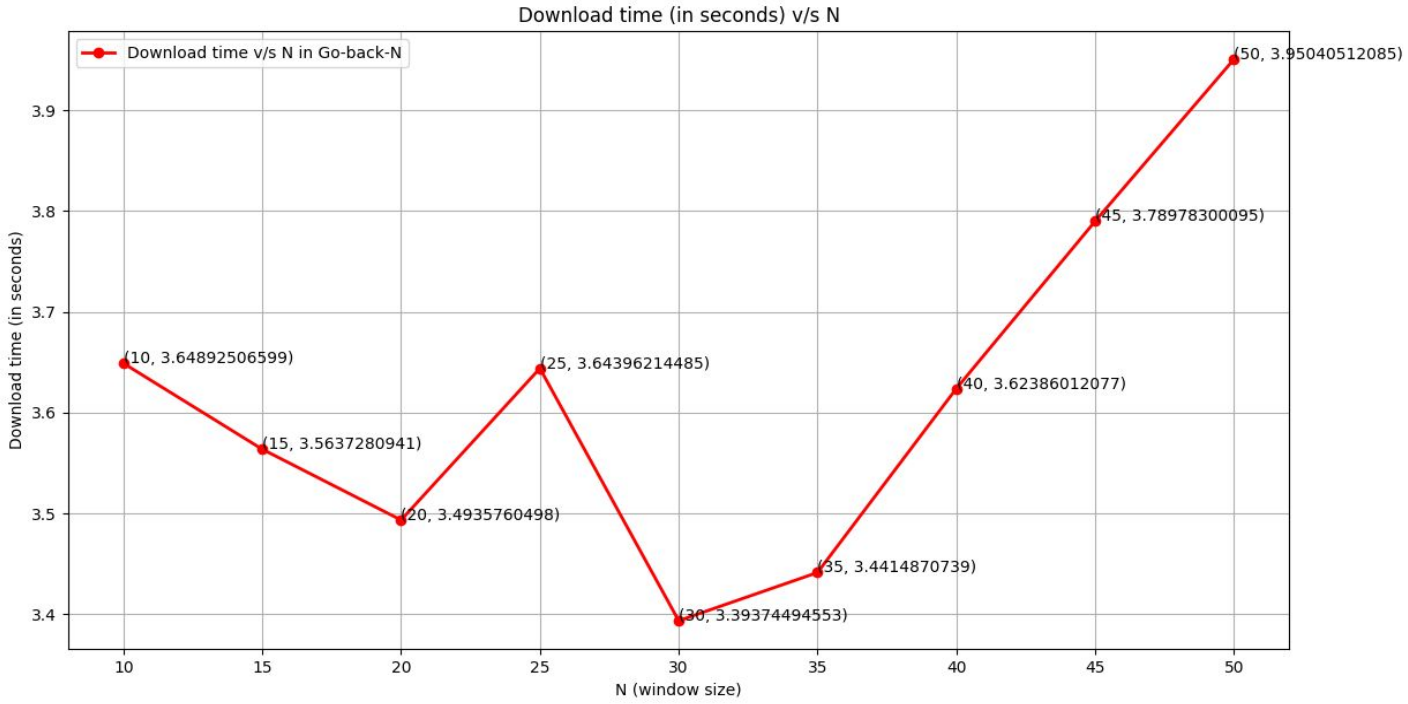
GBN-20 :



As a final observation in the last two cases, we observe that the download time reaches around 11 seconds, as compared to the 221 seconds taken in the Stop-and-Wait protocol download. Thus, we infer that not only Go-back-N is faster than Stop-and-Wait, but also, increasing window size with packet delay decreases download time substantially, as we see in the case of GBN-15 and GBN-20 downloads.

(c) Varying the fixed window size (in packets) of GBN from 10 to 50 in steps of 5, and plotting the time to download for a fixed packet loss rate of 10^-2 and delay of 100 msec :

Graph :



From the above graph, we infer that at first, for a fixed packet delay and packet error/loss rate, the download time decreases with the increase in the window size and but increasing the window size after N = 30, then tends to increase the download time. This happens because the the window size increases, even if one the packet in the window being sent get lost or corrupted, the entire window then ends up being sent again, thus adding an overhead if the window size is too large. That's why it's advised to keep the window size in a Go-back-N protocol under a certain limit, depending on the network and connection conditions.

General inference :

We saw two methods of reliable application-layer protocols using the unreliable UDP transport protocol.

The first one was the Stop-and-Wait protocol, also known as the alternating bit protocol, where the sender sends one frame at a time and after sending each frame, the sender doesn't send any further frames until it receives an ACK from the client. After receiving a valid packet from the server, the receiver sends an ACK. If the ACK does not reach the sender before a certain time, known as the timeout, the sender sends the same frame again.

The second was the Go-back-N protocol, or the selective window protocol, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver. This method is more efficient as compared to the Stop-and-Wait protocol, since it sends all the packets it can, thus utilizing the bandwidth much properly and resourcefully as compared to the former, and this method is fast as well.

In simple words, unlike waiting for an ACK for each packet as is the case in Stop-and-Wait protocol, the connection is still being utilized as packets are being sent. In other words, during the time that would otherwise be spent waiting, more packets are being sent. But this leads to a drawback as well since this method also results in sending packets multiple times – if any packet was lost or corrupted, or the ACK lost or corrupted, then that packet and all following packet in the window (even if they were received without error) will be re-sent, thus, prompting a need to keep a check on the window size, N, in the Go-back-N protocol.

Hence both the protocols have their own pros and cons, the Stop-and-Wait protocol is not much efficient since it waits for an ACK for each packet sent, and in case of the Go-back-N protocol, when the window size is too large, the number of packets in the pipeline grows and one packet error causes the retransmission of many packets unnecessarily. Hence, depending on the needs of the user, appropriate protocol should be employed.