Agathe Benichou and Frankie Klerer
Professor Sadovnik
CS305: Computer Networks
Due: April 9th 2017

Project 2: Implementing a Reliable Transport Protocol Report

Introduction

Packets moving throughout a network often face a wide range of difficulties. They are subject to loss, reordering, queue overflows at routers and switches as a result of congestion and routing failures and corruption. Applications, such as the Internet, need a reliable data transport protocol of which they can rely on to deliver their data in-order stream and with minimal casualties. However, these protocols cannot prevent the network from not affecting the packets. As a result, the protocols need to be able to handle any sort of loss, corruption and reordering that could happen within the network. Our program implements two such protocols: Go-Back-N and TCP. Go-Back-N is a stop and wait as well as a sliding window protocol in which the sender continuously sends packets within a designated window size frame without receiving an acknowledgement from the receiver. TCP is the protocol used by the Internet which provides reliable, ordered and error-checked delivery of a stream of bytes. Our program simulates a hardware and software environment while implementing a sending and receiver transport-level code for implementing these two reliable data transfer protocols.

Description of GBN and TCP

All reliable data transfer protocols must be able to provide a guaranteed connection-oriented data delivery service between the sending the receive process. Both Go-Back-N and TCP are able to cope with losses, reordering, corruption, congestion by using a timer as well as many other checks. While Go-Back-N and TCP are different protocols, they achieve the same task. These reliable data transport protocols both run in the end systems such as

clients and servers. On the sending side, the application layer passes messages to be sent down to the transport layer. The transport layer converts these received messages into a transport layer segment with a header. The transport layer passes the segment to the network layer, where it is encapsulated into a datagram to be sent across the network to the receiver, and the reliable data transport protocol ensures that the message is properly received by the receiver. Once the receiving sides network layer receives the datagram, it is extracted and passed up to the transport layer where it is processed and then sent to the application layer where it is accessible by the receiver.

Both protocols use a variety of methods to identify packets, track packets and ensure that all packets will be sent in order to the receiver. Before one's application layer can begin to send data to another, the two processes must first handshake to ensure the connection. This includes sending some preliminary segment to each other to establish the parameters of the ensuing data transfer. Both protocols are full duplex which means data can flow from A to B and from B to A simultaneously. Every packet that is to be sent to the receiver is given a packet number, or a sequence number, which is always in order. The first packet to be sent is given packet #0, the second packet to be sent is given packet #1, etc. The receiver uses these sequence numbers to track the packets it has received. When the receiver has correctly received a packet, the receiver sends an acknowledgment back to sender indicating that it has received the packet. These ACKs are also given numbers but these numbers depend on the protocol.

Go-Back-N (GBN) is not used in the world today but it is nonetheless a sturdy protocol that achieves its purpose. GBN is known as a sliding window protocol so it would make sense that the GBN sender has a window size. At any given moment, the window contains four different types of packets: a packet that has been sent into the network and acknowledged by the receiver(telling the sender that it has properly received the packet), a packet that has been sent into the network but not yet acknowledged by the receiver, an open slot in the window for a packet to be sent and future packets that cannot be dealt with because there is not enough room in the window to process it. This is well depicted in Figure 1. The window size restricts the

sender from only having up to N unacknowledged packets in its pipeline where N is the window size.

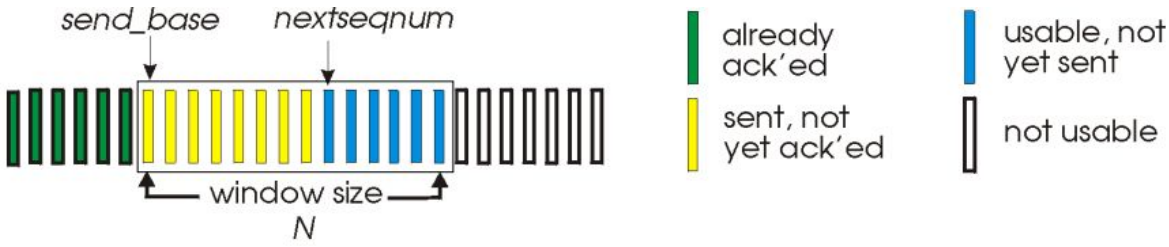


Figure 1 (from textbook): Sender window at any moment in time

The sender will respond to three different events while sending packets: invocation from above, receipt of an ACK and a timeout event. When the application layer on the sender's side passes a message to the transport layer, the sender must check to see if there is room in the window to send the packet. If the window is not full, the packet is added to the window, sent into the network and marked as sent but not yet ACKed. If the window is full, the sender returns the message to the application layer so that the application layer knows that it can try to send the message later. The sender keeps a timer for the oldest packet in the window that has not yet been acknowledged. The timer limit is set by the network administrator. When the timer for the packet expires, the sender retransmits all packets in the window that has been sent but has not been acknowledged. This means that even if there other packets in the window are successfully being transmitted, the sender will retransmit them into the network. The receiver waits to receive packets and when it does, it transmits an acknowledge message (ACK) to the sender to let it know that it has properly received its message. The receiver is able to send cumulative ACK to the sender so that if the receiver has properly received packets 1, 2, 3, it sends an ACK for packet 3 and the sender knows that the receiver has properly received all packets from 3 and below. However, if the receiver has received packets 1 and 3 but not 2, it will discard packet 2 because it is out of sequence. The receiver will not buffer packet 3 until it receives packet 2, instead it sends an ACK for the most recently received and ACked packet which is packet 1 which tells the

sender that it is still waiting for the packet after 1, so packet 2. Accordingly, the receiver always has a window size of 1 so it is only concentrated on receiving the next in order packet. If the receiver correctly receives a packet, say packet 1, whose sequence number is in order, then the receiver sends an ACK 1 for packet 1 and delivers the data of the packet to the application layer. If a packet gets lost while trying to reach the receiver, the receiver will discard any packets whose sequence number is above the lost packet and transmit an ACK for the highest in order ACK. If an ACK gets lost on its way to the sender, the sender will wait until the timer expires and retransmit all packets in the window.

TCP is the Internet's transport layer, connection oriented, reliable transport protocol. TCP creates a reliable data transfer on tops of IP's “best effort” delivery service which ensures that a data stream is uncorrupted, in order and without gaps or duplications.The sender application passes a stream of data to the transport layer where it is now in the hands of TCP. TCP directs this data to the sender's send buffer so that TCP can grab data from the sender buffer to send to the receiver. Every chunk of data is paired with a TCP header to form TCP segments. When the TCP receives a segment at the other end, the segments data is placed into the receive buffer. The applications transport layer will take data from this receive buffer and pass it to the application layer. A visual image of this process can be seen in Figure 2.

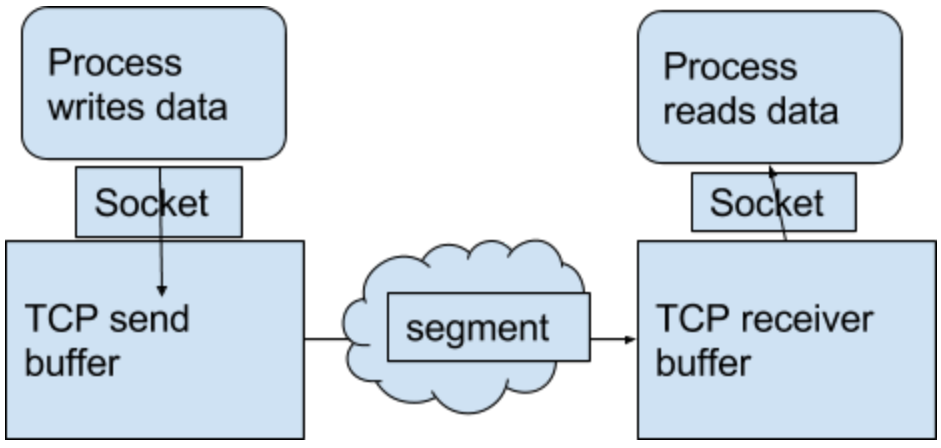


Figure 2: Process of sending a packet through the network

TCP views data as an unstructured but ordered stream of bytes. The sequence number for a segment is the byte stream number of the first byte in the segment. Additionally, the ACK number that the receiver sends back is the packet that it is expecting next. So if the sender sends packet 1, the receives replies with ACK 2 and the sender know that the receiver has correctly received packet 1 and that it is now waiting for packet 2. The TCP sender must respond to three different events: invocation from above, timer timeout, ACK receipt. When TCP receives data from the application above, it encapsulates the data in a segment and passes the segment to the network IP. TCP uses only a single retransmission timer which means that every packet in the window has its own timer. When the timer expires, the sender retransmits the smallest, unACKed packet. Upon the receipt of an ACK, the sender makes sure that the received ACK number corresponds with the packet number it is about to send. The receiver waits to receive packets and when it does, it transmits an acknowledge message (ACK) to the sender to let it know that it has properly received its message. The receiver is able to send cumulative ACK to the sender so that if the receiver has properly received packets 1, 2, 3, it sends an ACK for packet 3 and the sender knows that the receiver has properly received all packets from 3 and below. If an out of order packet arrives at the receiver, it is buffered until all packets before it have been received. The

If an ACK gets lost on its way to the sender, the sender waits until the timeout for that packet occurs and then it retransmits the packet again. TCP utilizes a method called Triple Duplicate ACK so that if the TCP sender receives 3 duplicate ACKs for a packet that has already been ACKed, it takes it as an indication that the following packet has been lost. From the sender's perspective, if we send packets 1 2 3 4 5 6 and get back ACK 1 2 2 2 2 then packet 2 is lost which is why we got stuck at ack 2 and packets 4 5 6 probably did make it and triggered the three duplicate ACK 2. TCP only retransmits at most one packet.

However, not all is perfect in the network. It is very easy for packets to be corrupted which affects their sequence number, ACK number and the message they are carrying. GBN and TCP use a method called checksum which detects errors by adding together all the bytes in the packet and storing the sum in the header. The sender performs checksum on the packet before

sending it out into the network and places the sum value in the header. Upon receiving the packet, the receiver also performs checksum and compares the value calculated with the value stored in the header. If the values match, the receiver proceeds by sending an ACK and delivering the packet to the application layer. If not, then the receivers actions change a little. For GBN, when the receiver receives a corrupted packet, it sends an ACK for the packet with the highest sequence number that has already been ACKed. When the sender receives an ACK for a packet whose ACK it has already received, the sender knows that something went wrong and it retransmits all the packets in the window. For TCP, when the receiver receives a corrupted packet, it sends an ACK for the highest sequence number that has already been ACKed. When the sender receives an ACK for a packet whose ACK it has already received, it knows to resend the packet that comes immediately after that one.

There are many advantages and disadvantages to both TCP and GBN. Some advantages to GBN include: the sender being able to send many packets at any one time, the timer is set for a group of packets and that one ACK can acknowledge multiple packets. Some disadvantages to GBN include: the unnecessary retransmission of all the packets in the window frame, the sender needs to retain its messages until an ACK has been received for them all and the receiver does not buffer out of order packets. Some advantages to TCP include: guaranteed delivery service without duplication, no unnecessary retransmissions and cumulative ACKing. The only disadvantages of TCP are network related issues such as it cannot be used for broadcast and multicast connections.

Code Design

The structure of our program and class design is shown in Figure 2. The overarching Network Simulator class mimics the interaction between two end systems and initializes the connection via a 3-way handshake. It creates the Sender Application class which is the application layer for the sending side which is used in the creation of the Sender Transport class which is the transport layer for the sending side. It also creates the Receiver Application class

which is the application layer for the receiving side which is used in the creation of the Receiver Transport class which is the transport layer for the receiving side. These two “end systems” are connected via one Network Layer class which simulates the network environment between the sending and receiving side.

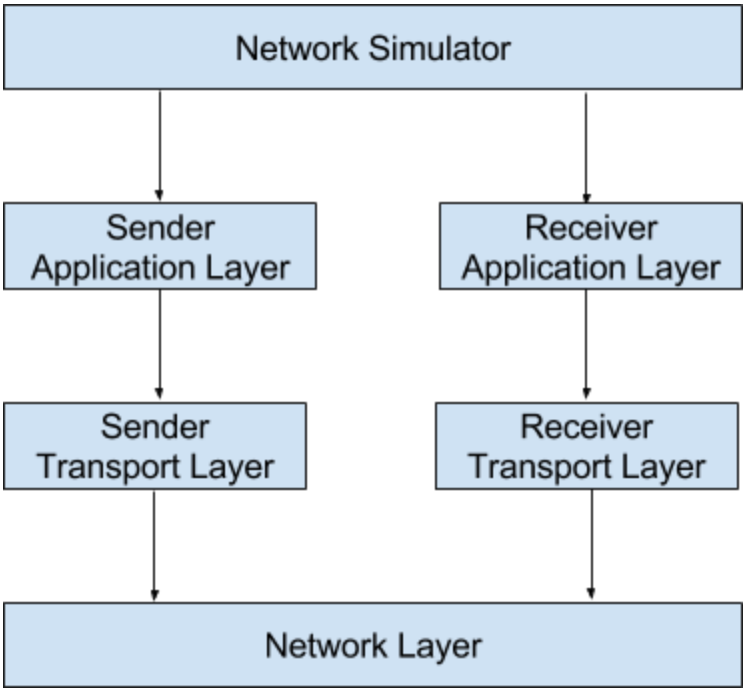


Figure 3: Structure of our program

The Network Simulator class reads in the programs parameters, parses it and distributes the appropriate parameters accordingly. The Sender Application class contains the messages in an ArrayList of Message objects to be sent to the Sender Transport class. The Sender Application class sends one message at a time to the Sender Transport class which envelopes the message into a packet and processes it accordingly. The Sender Transport class is dense and uses many data structures to process the message. There is an ArrayList which records the status code of the packets and updates it throughout the simulation. The index of the array corresponds to the packet sequence number and it has different number codes where 1 means the current window but not yet sent, 2 means sent but not ACKed, 3 means sent and ACKed, and 5 means ready to send but not space in window. To keep track of the packets, the class also contains a current window ArrayList which is set to the size of the window size and keeps track of the packets that

are currently in the window. There is a hashmap which contains the Packet objects that have been sent into the network where the key is the packet number and the value if the Packet object itself. There is an ArrayList which stores the messages that the Sender Application class tried to send but failed because the window was full. There is an ArrayList which stores the amount of times a packet has been ACKed, this is mostly used for TCP to detect triple duplicate ACKs. The Sender Transport class sends messages for TCP and GBN the same way: for every packet in the packet list, if the packet has not yet been sent and if its sequence number is the next one expected to be sent out, create two instances of the Packet object where one is stored into the HashMap and the other is sent into the network. The packet status code ArrayList is updated, the packet is added to the current window and the timer is set (if it hasn't been set before).

The Sender Transport class sends the message to the Network Layer class where it has the chance of being corrupted but delivers it to the Receiver Transport class. There is an ArrayList which records the status code of the packets and updates it throughout the simulation. The index of the array corresponds to the packet sequence number and it has different number codes where 1 means the packet has not yet been received and 2 means the packet has been received and ACKed. To keep track of the packets, the class also contains a current window ArrayList which is set to the size of the window size and keeps track of the packets that are currently in the window. There is an ArrayList in charge of keeping track of all the buffered packets (for TCP) that have been received out of order. When the receiver receive a message from the network layer, it does things different depending on if it is using TCP or GBN. If it's using TCP: it checks to see if the packet is corrupt. If the packet is corrupt, then it sends an ACK for the packet after the highest in order packet that has already been HACKed since the receiver for TCP sends an ACK for the packet it is expecting. If the packet is not corrupt, it checks to see if there is a packet before this one that has not yet been ACKed. If there is, it buffers the received packet and sends an ACK for the highest in order packet that has already been ACKed. Upon every arrival of a packet, the program checks if it can update the buffer. It finds the highest sequence number in the buffer list and it checks to see if all packets before that one have been received. If they all have been received, it sends them all to the Receiver Application class. If all

packets before that have not yet been received, it moves on since there is nothing it can do. If there is no gap in the order of received packets, then if you received a packet that has not yet been ACKed, process the packet to the Receiver Application class. If the received packet has already been ACKed, resend the ACK for the highest in order packet since there is something out of order in the sender. If it's using GBN: it checks to see if the packet is corrupt. If the packet is corrupt, then it sends an ACK for the highest in order packet that has already been ACKed. If the packet is not corrupt, it checks to see if there is a gap between the first packet and the packet it has just received. If there is a gap, it discards the received packet and sends an ACK for the highest in order packet that has already been ACKed. If no gap is detected, then if the packet has not yet been ACKed, send the sender an ACK and deliver it to the Receiver Application class. If the packet has been HACKed already, resend the ACK for the highest in order packet since there is something out of order in the sender. The Receiver Application class just prints out to the program that it has received a packet.

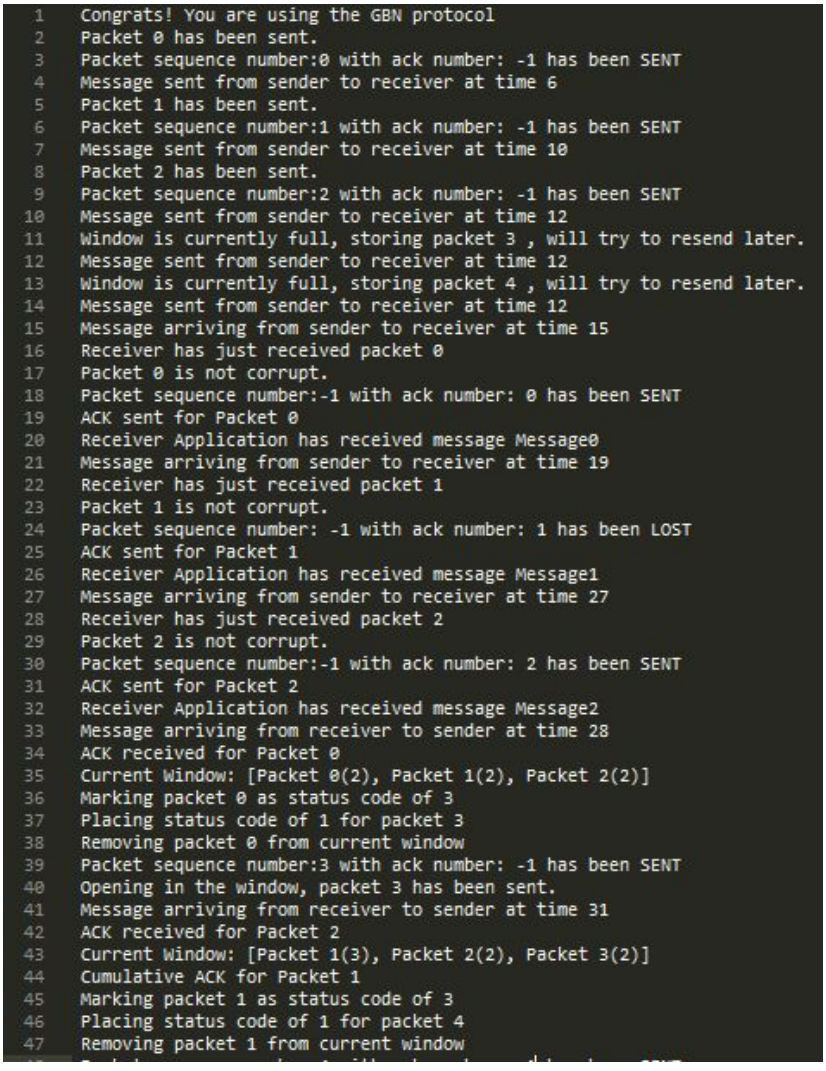
In any case, the Receiver Transport class sends the Sender Transport class an ACK for the packet. The Sender Transport class must process the received ACK. If the program is using TCP, it must check if the received ACK is corrupt. If it corrupt, it does not do anything so the program will just wait until the timer for that packet times out. If it is not corrupt, it checks to see if the received packet has already been HACKed or if the packet has not yet been ACKed. If the packet has already been ACKed, it increments the ACK count in the ArrayList and checks to see if the ACK count has reached 4. If the packet has not yet been ACKed, it checks to see if there are any packets before it that have not yet been ACKed. If there are, then it cumulatively ACKs all the packets and moves the sender's window. If the program is using GBN, it must check if the received ACK is corrupt. If it is corrupt, it resends all the packets in the current window. If the packet is not corrupt, it checks to see if the first packet was lost or not and acts accordingly. If the first packet wasn’t lost, and if the ACK for that packet has already been received then resend all the packets in the current window. If the first packet wasn’t lost and if the ACK for the packet has not yet been received, it checks to see if there are any packets before the received packet that have not yet been ACKed. If there are, then it cumulatively ACKs all the packets before it and

moves the sender's window. When the timer expires for TCP, it retransmits the smallest unACKed packet. When the timer expires for GBN, it retransmits all the packets in the current window.

Results

To test our program for its correctness, we tested it with 5 messages, a 0.1 probability for loss , a 0.1 probability for corruption and a window size of 3.

The trial for GBN can be seen in Figure 4:



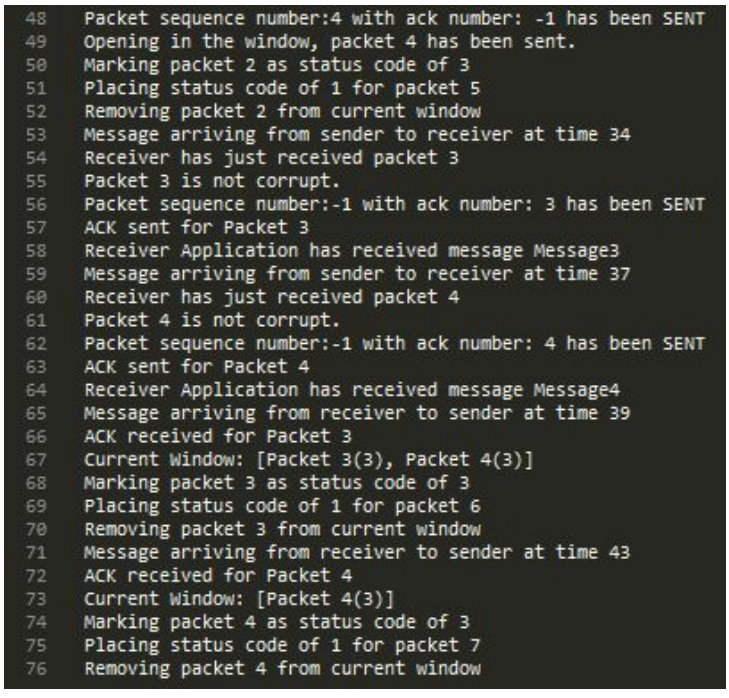


Figure 4: Printout from GBN Trial

For this trial, the ACK for packet 1 was lost (line 24) but the sender received the ACK for packet 2 before the timer expired so packet 1 was cumulatively ACKed (line 44). The corresponding sending diagram is shown in Figure 5.

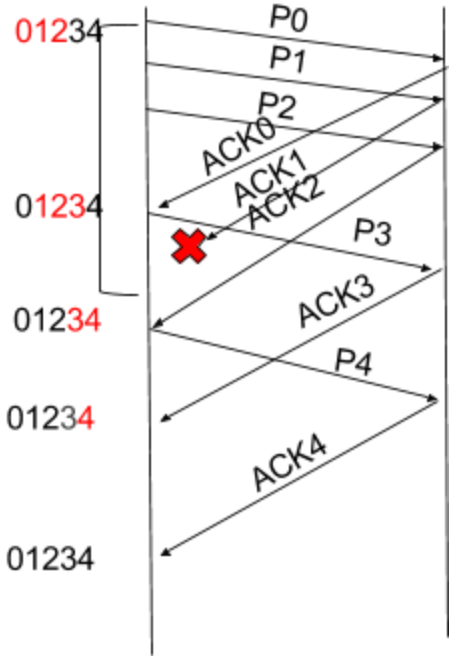


Figure 5: Sending diagram for GBN trial

Since the time it takes to transmit a message from sender to receiver, or vise versa, is about 10 simulated seconds on average. A test with 5 packets, and since the loss event does not cause a retransmit due to cumulative ACK's. The first 3 packets would take RTT + 3L/R. L/R is negligible in this situation because we are sending extremely small packets, but the time in between each packet is significant. Packet 2 gets sent at time 10. The fourth packet gets sent while the sender is waiting for packet 1 and 2. Then when the ACK for 2 is received the last packet is sent. One RTT is 20 simulated seconds. So according to this test, the total running time should be about 2*RTT + 10. This equals 50, similar to our simulation which returns a time of 43 simulated time units.

The trial for TCP can be seen in Figure 6:

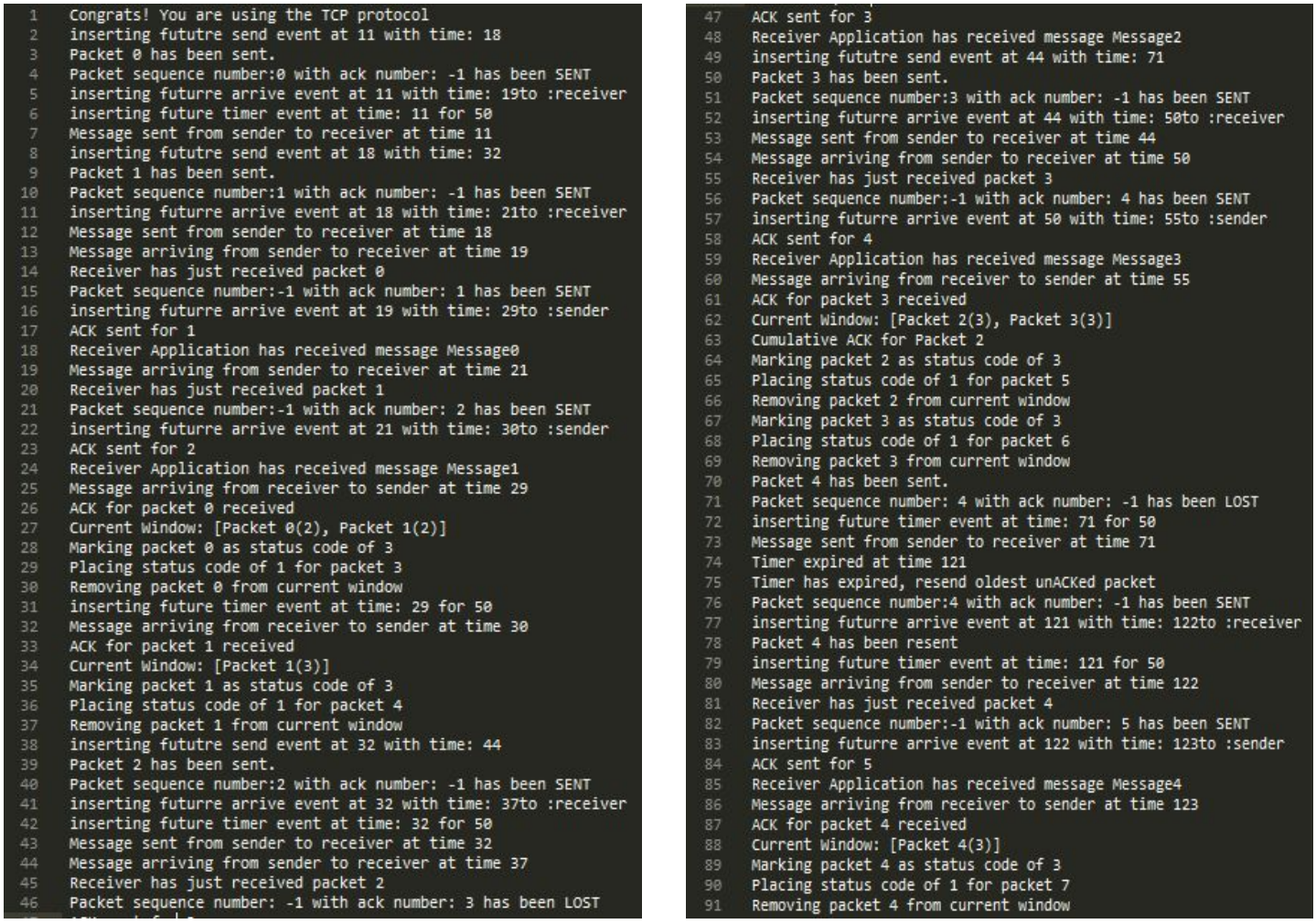


Figure 6: Printout from TCP Trial

For this trial, the ACK for packet 2 was lost (line 46) but the sender received the ACK for packet 3 before the timer expired so packet 2 was cumulatively ACKed (line 63). On line 71, packet 4 gets lost but resent when the timer expires on line 75. The ACK for packet 4 was received by the sender on line 87. The corresponding sending diagram is shown in Figure 7.

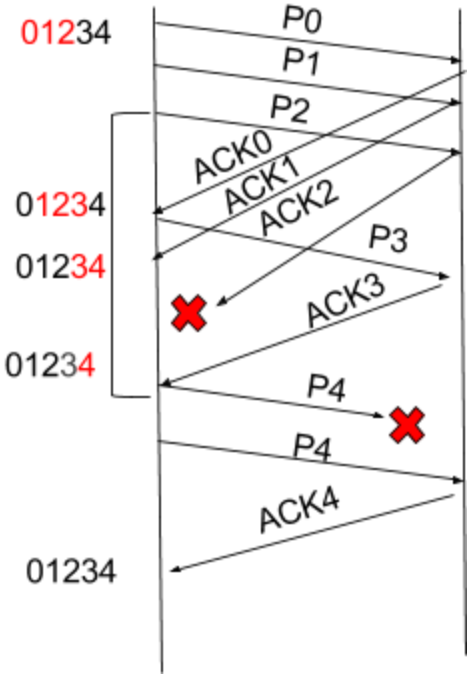


Figure 7: Sending Diagram for TCP Trial

Again, the time it takes to transmit a message from sender to receiver, or vise versa, is about 10 simulated seconds on average. This makes the RTT equal to 20 simulated seconds. Transmission time from the sending of packet 0 to the received ACK of packet 3 is equal to 2*RTT, since only one ACK is lost, which then gets cumulatively ACKed before a timeout. Then packet 4 gets sent and lost, the timer is set for 50 simulated seconds, 50 seconds later, packet 4 gets resent and another RTT elapses. This totals to, 3*RTT + 50 = 110 simulated seconds, similarly, our test takes 123 simulated seconds.

Result Analysis

We analyzed several different aspects of our program but we kept the window size constant at 3 with 10 messages being sent in every trial.

Figure 8 shows the comparison of GBN and TCP over the increasing probability of packet loss. Starting at a packet loss of 0, GBN and TCP perform at about the same speeds. As packet loss increases to 0.2, 0.4 and 0.6 probability of losing a packet, the time it takes TCP to process all 10 packets continues to exceed the time it takes GBN to process all 10 packets. This might be due to the fact that GBN retransmits all packets in its window upon a packet loss so the chances of one of those packets or ACKs being received correctly is higher. For example, if the sender retransmits packets 1, 2, 3 and the ACK for packets 1 and 2 gets lost, then the ACK for packet 3 still tells the sender that packets 1 and 2 have been properly received. As the packet loss probability increases, the time for TCP increases more than the time for GBN. GBN takes less time during packet loss, because each time out, all the packets in the window get resent. More data gets sent, but the overall time is less than TCP. TCP will only re-send single packets on timeouts. For realistic implementation, TCP is better though because it sends less data over the network.

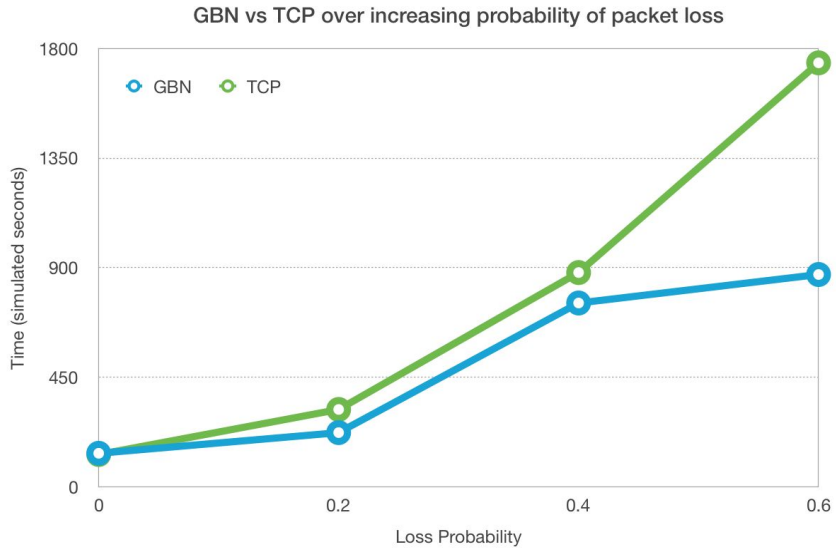


Figure 8: GBN vs TCP over increasing probability packet loss

Figure 9 shows the comparison of GBN and TCP over the increasing probability of corruption. Starting at a packet loss of 0, GBN and TCP perform at about the same speeds. As packet loss increases to 0.2, 0.4 and 0.6 probability of corrupting a packet, the time it takes GBN to process all 10 packets continues to exceed the time it takes TCP to process all 10 packets. This might be due to the fact that GBN retransmits all packets in its window upon a packet loss so when one packet or ACK is corrupt, and since the GBN receiver always has a window size of 1, and does not buffer out of order packets, it must wait to get all packets in order. Causing many retransmissions if the corruptions probability is high. In comparison, TCP buffers the out of order packets and sends a duplicate ack for the missing packets that may have been corrupted, and when a packet gets three acks it retransmits. This causes less unnecessary retransmission, and efficient resend method of the corrupted packets.

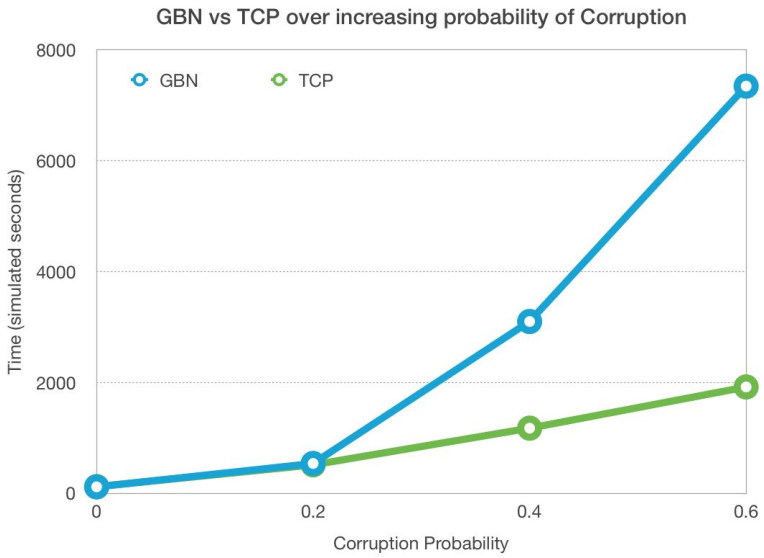


Figure 9: GBN vs TCP over increasing probability of corruption

Figure 10 shows the comparison of GBN and TCP over the increasing probability of packet loss and corruption. Starting at a packet loss of 0, GBN and TCP perform at about the same speeds. As packet loss increases to 0.2, 0.4 and 0.6 probability of losing a packet, the time it takes TCP to process all 10 packets continues to exceed the time it takes GBN to process all 10 packets. It takes TCP longer to recover from packet loss and it takes GBN longer to recover from corruption. These two factors combined with an increasing chance of loss and corruption affect TCP more than it affects GBN. As we've shown, GBN re transmits many more packets, so more data is being sent, but with increasing chances of loss and corruption, GBN's many packets give a greater chance that some are not corrupted or lost. In TCP's case, it transmits single packets that were lost, so for each packet gets lost or corrupted again, you have to wait until a response from the receiver or a timeout.

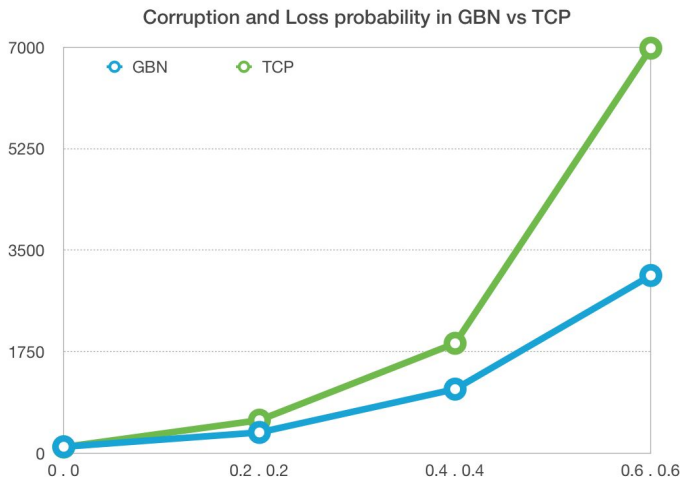


Figure 10: GBN vs TCP with mixed loss and corruption

Figure 11 shows the comparison of GBN and TCP over the increasing time between when the sender application layer passes messages down to the sender transport layer. TCP and GBN handle the increment about the same with GBN taking longer to process the 10 messages than TCP with a time delay of 25 seconds. Overall, with the no packet loss or corruption, TCP and GBN have similar running times.

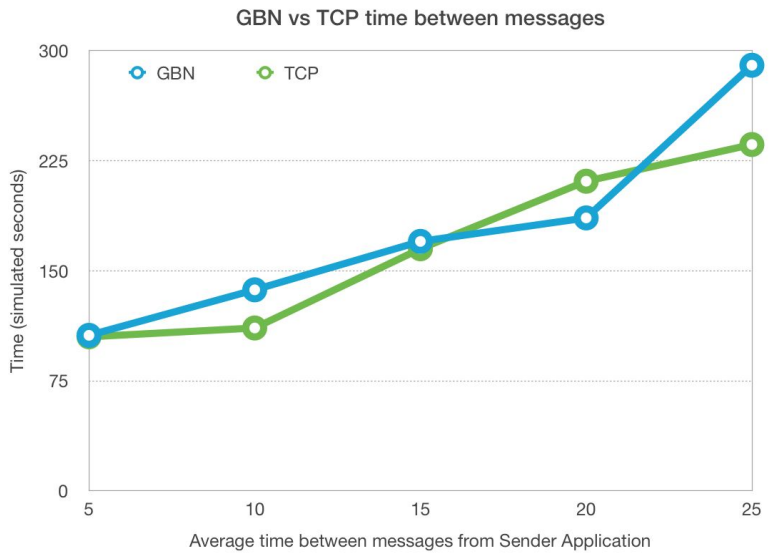


Figure 11: GBN vs TCP over increasing time between messages

Figure 12 compares the time it takes GBN vs TCP to process 10 messages while increasing the window size from 0 to 10 with no loss or corruption. With a window size of 1 until a window size of 8, GBN takes a longer time to process the messages than TCP. However, with a window size of 8, both TCP and GBN take the same time to process messages. With a window size of 10, which is the amount of messages being sent, GBN is slightly faster than TCP. Overall TCP and GBN deal with varying window size in the about the same time. Neither of the processes takes significantly more time.

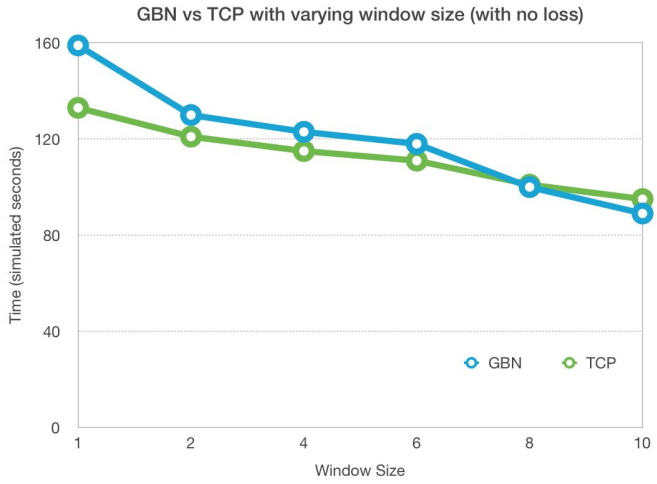


Figure 12: TCP vs GBN with varying window size

Conclusion

After testing TCP and GBN under a wide range of conditions and analyzing the results, it can be concluded that TCP works better for corruption and GBN works better for loss. The triple duplicate ACK avoids unnecessary retransmission so TCP is able to avoid congestion in the network. GBN cannot buffer out of order packets, it must discard out of order packets until it has received the next in order packet. In this case, many transmissions allow packet loss to be avoidable. As a result, GBN works better for loss.

References

Kurose, James F., and Keith W. Ross. *Computer Networking: A Top-down Approach*. N.p.: Pearson, 2017. Print.

"Transport Services and Protocols." *Data Networks, IP and the Internet* (n.d.): 277-315. *MIT*. MIT. Web. 9 Apr. 2017.

Sharma, Praneet. "ADVANTAGES AND DISADVANTAGES OF TCP AND UDP."ADVANTAGES AND DISADVANTAGES OF TCP AND UDP. N.p., 01 Jan. 1970. Web. 09 Apr. 2017.

Contributions

As a team, we contributed equal parts to the code as well as the report.