ECE358 - Project 2
Data Link Layer and ARQ Protocols

Objective:

After this experiment, the students should understand:

i. How different retransmission (ARQ) protocols work.

ii. The effect of different parameters on the performance of the protocols, including channel capacity, propagation delay, size of data frame, size of ack frame, and error ratio.

Programming languages: this experiment can be done using C, C++ or Python.

1. Overview:

In the OSI model, the *data link layer (layer 2)* handles data transfer over a single, bi-directional communication link without relays or intermediate nodes. The link can be point-to-point, broadcast, or switched. The data link layer is made of 2 components. The lower one is the MAC (Multiple-Access Control) sub-layer that manages multipoint links (this means that if the link is point-to-point, there is no MAC protocol). The upper one is called LLC (Logical Link Control). This project focuses on LLC and hence assumes a point-to-point link. On the sending side the data link layer encapsulates protocol data units (PDUs), also known as packets, coming from the network layer (layer 3) into a layer 2 PDU called a frame, adds its header including error control bits (e.g., CRC), and sends the whole frame to the physical layer (layer 1). The data link layer supervises the physical transmission of frames by the physical layer, performs error control over the link (at least error detection), and possibly initiates retransmission when errors are detected and cannot be corrected. Therefore, in its most complete and complex form, the data link layer provides *a reliable point-to-point* communication service between two adjacent nodes, with each corresponding layer 2 peer process capable of sending and receiving frames. See Figure 1.

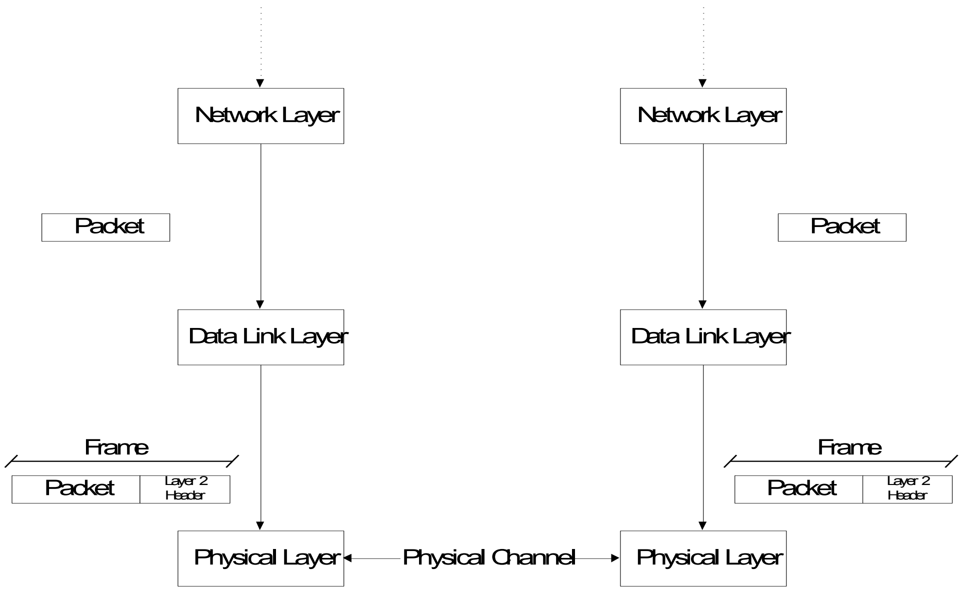


Figure 1 Relationship between packets and frames

There are two main types of error control: *Forward Error Correction (FEC) and Automatic Repeat reQuest (ARQ)*. FEC will embed *error correction code* into the frame and the receiver will try to correct the errors, while ARQ will embed *error detection code* in the frame and the receiver will detect errors and request retransmission if errors occur. Although ARQ involves extra overhead for retransmission requests and retransmitting frames that were received with errors, the total overhead for data links with reasonable error rates is often far less than it would be with FEC.

It is interesting to note that ARQ is not only applicable to the data link layer, but also to the transport layer and hence what you are learning here is crucial for your understanding of TCP. The main difference between an ARQ protocol at layer 2 and layer 4 is that at layer 2, the sender and the receiver are connected through a physical channel which can corrupt, delay or lose frames while at layer 4, the sender and the receiver are connected through a network which can not only corrupt or lose layer 4 PDUs (called segments) but also introduce more delay and delay variation than a channel and deliver segments out-of-order. In the following, we only consider the case of layer 2 ARQ protocols.

2. Background Material:

In this experiment, we will investigate two different ARQ protocols, namely, *Alternating Bit Protocol (ABP)* and *Go Back N (GBN)*. We will focus on one direction for data transmission. In general data flows in both directions. Referring to Figure 2, A is the side sending data and B is the side receiving it and sending acknowledgements (ACK).

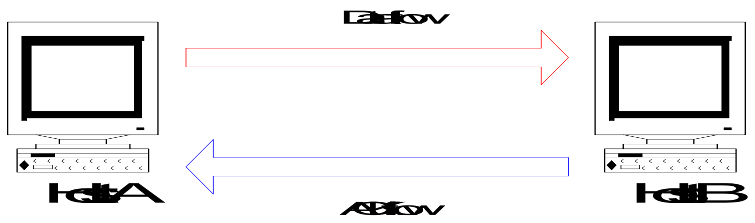


Figure *2*

2.1 ABP

In ABP, the sender (sender here refers to the data link layer process) sends packets in data frames numbered 0 and 1 (this number is called a *sequence number*) alternatively and starts a timeout after each frame it sends. (The frame is called a data frame because it actually contains data from layer 3). The sender has a buffer able to contain 1 packet in which the packet currently being sent is kept until the sender is sure that it has been received correctly by the other side. The sender sends one packet at a time, i.e., it does not send packet #j before being completely sure that packet #(j-1) has been correctly received. When the sender has completely finished with packet #(j-1), i.e., it has received a non-corrupted ACK for it, it empties its buffer and requests a new packet from the upper layer. The upper layer may not have a packet to send right away but is then aware of the availability of the layer 2 sending process.

The channel can corrupt or lose a data frame or an ack frame. The receiver keeps a counter NEXT_EXPECTED_FRAME which represents the frame sequence number that it is expecting. A correct frame for the receiver is a frame that is received without error. If the receiver receives a correct frame whose sequence number is equal to NEXT_EXPECTED_FRAME, it interprets it as a new frame with a new packet. It sends the packet to Layer 3, increases its counter NEXT_EXPECTED_FRAME by 1 (modulo 2) and acknowledges the receipt of the frame by sending an acknowledgement (ACK) which is encapsulated in a frame from B to A. This frame can either be a data frame from B to A or a special frame that only contains the ACK. We will assume the latter in the following. This ACK frame carries an acknowledgement number RN equal to the updated value of NEXT_EXPECTED_FRAME. If the receiver receives a frame in error or a correct frame with an unexpected SN number (i.e., not equal to NEXT_EXPECTED_FRAME), it will not update the counter value NEXT_EXPECTED_FRAME and will send an ACK with the current value of NEXT_EXPECTED_FRAME as RN.

The sender maintains a counter NEXT_EXPECTED_ACK which is set to SN+1 (modulo 2) where SN is the sequence-number of the latest frame sent by the sender. It will wait for the receiver's ACK with the right ACK number (i.e., RN being equal to NEXT_EXPECTED_ACK) before emptying its buffer of the current packet, increasing SN and NEXT_EXPECTED_ACK by 1 (modulo 2) and informing the upper layer that it is ready to send a new packet. It will send a new packet in a new numbered frame as soon as the layer 3 of A has one to send. More precisely, when it receives a frame, if it is correct, it checks the value of RN. If RN = NEXT_EXPECTED_ACK, it does what was explained earlier in this paragraph, otherwise if the frame is correct but RN is not equal to NEXT_EXPECTED_ACK or if the frame contains errors, 2 options are possible. In the first option, the sender does not do anything, while in the second option, it resends the packet in the buffer in a new data frame with the same SN. If the sender does not receive an acknowledgement before the frame's timeout, either because the ACK was not sent by the receiver or was lost or because the timeout was too short, the sender will retransmit the same packet in a new frame carrying the same sequence number SN as the original frame.

2.2 GBN

Go Back N works in a more efficient way. With a given *window size* N, the sender will be able to deal with more than one packet at a time. It will number the frames sequentially modulo (N+1), i.e., 0, 1, 2, …, N, 0, 1, 2, … N. The window size corresponds to the maximum number of frames that can be in flight, i.e., sent sequentially without waiting for an acknowledgement. The sender has a buffer which is able to contain N packets and keeps the packets being sent till they are acknowledged in the order they have been sent. The sender has a pointer P that points to the frame sequence number of the oldest packet not yet acknowledged or to NIL if there is no such packet. The sender can send at most N packets in frames numbered P, …, P+N-1 (modulo N+1) before receiving a correct acknowledgement for frame numbered P (i.e., a RN number equal to P+1). Sometimes, the upper layer does not have a packet to send in which case there will be less than N packets in the buffer. We will assume that Layer 3 always has packets to send and hence the buffer always contains N packets. The value of pointer P can only be updated (i.e., we can only slide the window) when frame numbered P has been acknowledged. The new value of P is not always P+1 modulo N+1 since acknowledgements are cumulative and ACKs can be lost. We will describe the window-update in more details after we detail the behaviour of the receiver.

The receiver is very similar to the ABP receiver. The receiver keeps a counter NEXT_EXPECTED_FRAME and will only consider a frame with sequence number equal to NEXT_EXPECTED_FRAME as a new frame and send the encapsulated packet to Layer 3. It will update NEXT_EXPECTED_FRAME by incrementing it by 1 (modulo N+1) and acknowledge the correctly received frame by sending an ACK with RN = NEXT_EXPECTED_FRAME (i.e., the updated value of the counter). This acknowledges all packets older than the one sent in frame with SN = RN-1 (modulo N+1).The receiver will also acknowledge every correctly received (out-of-order) frame by sending an ACK with RN = NEXT_EXPECTED_FRAME (without updating it first). Hence if the receiver is expecting frame 3 (i.e., it has received all frames up to 2 correctly) and correctly receives frame 4, it acknowledges it by sending an ACK with number 3 (which means that it has correctly received all frames up to 2 (inclusive)), and is expecting 3. The receiver would not keep the packet contained in frame 4 but would discard it. Note that this behaviour is exactly like that of the ABP receiver, except the receiver employs a modulo N+1 operation instead of a modulo 2 operation when incrementing its counter.

The GBN sender waits for an ACK from the receiver. Upon receiving an ACK with a given RN, the sender will be able to know whether *one or more* of its sent packets were received successfully by the receiver since by requesting a certain RN, the receiver implicitly acknowledges receipt of all packets older than the one that was carried in frame with SN = RN-1 (modulo N+1). If the channel was error-free, meaning no frame or ACK would ever be corrupted or lost, a GBN sender would receive an ACK with RN = P + 1 before receiving an ACK with RN = P + 2. But, if channel is not error-free (which is the case in practice), sometimes an ACK with RN = P + 1 could be lost or errored, and the sender can thus receive RN = P + 2, even though it is expecting RN = P + 1. Note that a sender can interpret the ACK with RN = P + 2 as a *cumulative positive* *acknowledgement* for both packets with SN = P and SN = P + 1. In other words, a GBN sender has *a set of expected RNs* for an upcoming ACK. In fact, with our assumption of the buffer always having N packets, any RN except RN = P can be considered as a positive acknowledgement. If the RN matches with any one of these *expected RNs*, the sender knows that all the packets older than the one sent in a frame with SN = RN-1 have also been acknowledged. It removes those packets from the buffer and fills the buffer with the same number of new packets. All new packets are then sent to the channel. This also updates the value of P, and is often referred to as *sliding the window*.

Like in ABP, the GBN sender also maintains a time-out of duration Δ. If t_P is the time at which the lowest numbered packet in the buffer is completely sent to the channel, a timeout event is set for $t_P + \Delta$. If the window does not slide by this time (because no right ACK was received by this time), a time-out event is said to have occurred. The sender responds to a time-out event by retransmitting all packets in the buffer (with the same frame sequence numbers) and by adding a new time-out event. These retransmissions are required because the receiver discards all out-of-order frames, so the sender cannot just send the supposed lost frame. Note that, during the event of the sender sliding its window, the value of P (the oldest unacknowledged packet) changes, which means that t_P also changes. Hence the sender has to reset the time-out event to $t_P + \Delta$, as soon as it slides the window. In other words, there is always only one time-out event in the ES and it has to be updated either when a retransmission occurs, or when the window slides.

We have not discussed the behaviour of the sender when an ACK arrives in error or when it has a wrong value of RN (i.e., equal to P). In this Lab, we will assume that the GBN sender does not act upon these 2 events.

Note that when the window size equals 1, GBN will reduce to ABP.

3. Simulation Framework:

Figure 3 shows the system that you are going to simulate:

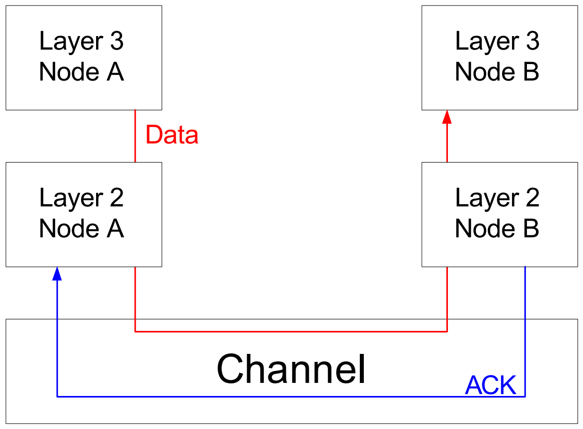


Figure 3 Model for simulation

We will assume that data always flows from node A (sender) to node B (receiver). You should bear in mind that in reality both nodes could be a sender of data and both could be a receiver of data. The channel is always bi-directional, even in this model, because data will go from A to B and ACK will go from B to A. Furthermore, we will assume that Layer 3 of node A ALWAYS has packets to send. Your simulator should be built based on all these assumptions.

We will use the DES (Discrete Event Simulation) framework introduced in Lab 1 to implement our simulator. A queue called the Event Scheduler (ES) will be used to store the events in sequence and a function will be used to return the event at the head of the event scheduler (the next earliest happening event), which we call the next event. The structure of an event has to be defined, with fields (properties) which are associated with it. Among other things, an event always has a *time* field, telling us when it occurred. There can be multiple ways in which the DES simulator can be designed. As a guideline, we will mention one particular approach and describe the important functional elements of the design. We expect you to follow this guideline. Fig. 4 shows the different functional components of this guideline design. Next, we will describe this design and the behaviours of the sender, receiver and the channel.

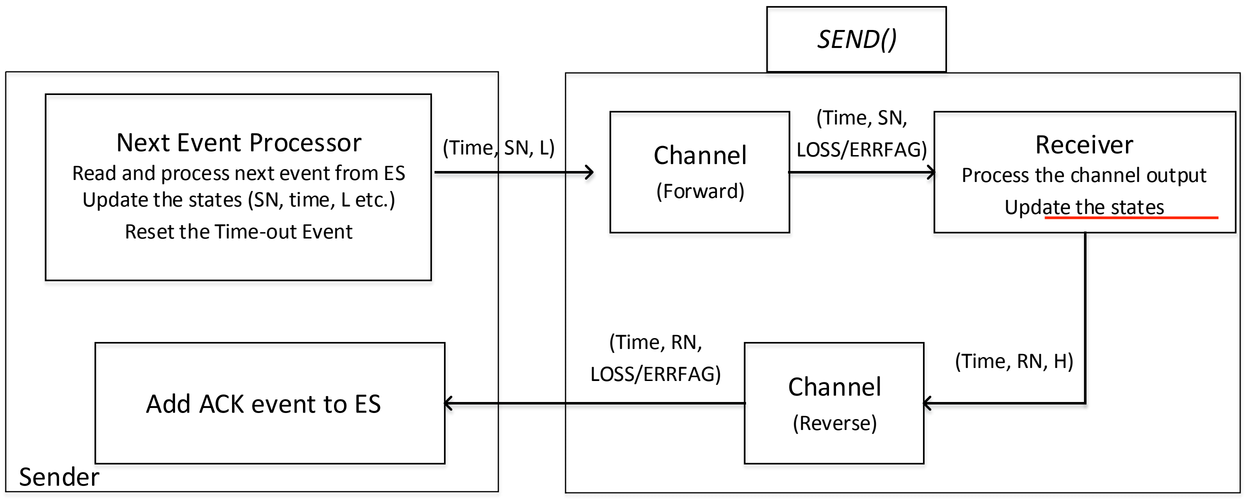


Figure 4: The functional diagram

First you need to define an event, which can be of one of two types: TIME-OUT and ACK. A TIME-OUT event has only one field: *time*. An ACK event has two extra fields: an associated RN, and its status, i.e., is it in error or not. So, you can define an event as a structure with four fields, namely *type* (TIME-OUT or ACK), *time, error-flag and sequence-number*. An event scheduler (ES) is a time-ordered sequence of such events. It is similar to the event scheduler that you designed in Lab 1.

We will briefly discuss the main components of the simulator, by first focusing on the ABP protocol. The details of event processing for GBN are discussed later.

Details of the ABP Simulator

1) **ABP Sender**: The sender maintains two counters, namely SN and NEXT_EXPECTED_ACK. SN is initialized to 0 and NEXT_EXPECTED_ACK is set to 1. In the beginning, the sender also chooses a value of time-out Δ which is kept constant throughout the simulation. The choice of Δ affects the protocol performance greatly. The *current time* t_c is initialized to 0. The sender generates a packet of length L = H + *l*, where H is the fixed length of the frame header and *l* is the packet length. Normally, packets have variable packet lengths. However, in this project, we assume that all packets are of the same length. The sender stores this packet in its (only) buffer. We assume (realistically) that the sender also knows the channel capacity C (in bits/sec). L/C is the amount of

time required for the frame to be completely sent into the channel, i.e., the transmission delay. The sender sends the packet in a frame with sequence number SN, which is completely transferred to the channel by time $t_c + L/C$, and at this point it registers a TIME-OUT event in the ES (i.e., a time-out at time $t_c + L/C + \Delta$). The sender then calls a function that implements *together* 1) the *forward channel* (from the sender to the receiver), 2) the *receiver* and 3) the *reverse channel* (from the receiver to the sender). Let this function be called *SEND()*. It returns as a value an event that has to be registered in the ES (See Fig. 4). Occasionally, a frame (on the forward channel) or an ACK (on the reverse channel) gets lost, in which case the function returns a NIL event which will not be registered in the ES. Otherwise, the return value of *SEND()* corresponds to an ACK event, which has a type: it can either come with or without error, and has a sequence-number RN. The sender reads the ES to get the *next event*. The reading process also updates the value of current time t_c, and removes the event from ES. If the next event is a TIME-OUT event (which means either the frame or the ACK was lost, or that the ACK arrived after the time-out), the same packet with the same SN is sent to the channel in a new frame. On the other hand, if the next event is an ACK without error and RN is equal to NEXT_EXPECTED_ACK, the sender knows its packet has been received correctly, so it increments SN and NEXT_EXPECTED_ACK by 1 (mod 2), generates a new packet, and sends a new frame. Note that there can be only one time-out in the ES, and hence as soon as the frame is completely passed on to the forward channel (i.e., at $t_c + L/C$), any outstanding time-outs in ES have to be purged and a new time-out at $t_c + L/C + \Delta$ has to be registered. Event processing takes non-zero time (for example, sending a frame takes L/C time), and we will assume that once the sender starts processing an event, it does not get interrupted by another event. The events are processed sequentially without interruptions.

2) The channel is characterized by its *propagation delay* τ where τ is assumed to be a constant quantity. A data frame (for the forward channel) or an ACK (for the reverse channel) can be lost with a probability P_{LOSS} (which, in general, can be different for a frame and an ACK). A lost frame/ACK never appears on the other end of the channel. With a probability $(1-P_{LOSS})$, the frame/ACK arrives at the other end of the channel, with two possibilities: a) it arrives in error (frame ERROR), b) it arrives without any error (frame NOERROR). These three events (frame LOSS, frame ERROR and frame NOERROR) are mutually dependent and usually have intricate relationships. To simplify the relationships among these outcomes, we assume that the channel has

a given bit error rate BER (the probability that a bit arrives in error, considered independently from one bit to another) and we *arbitrarily* consider a frame experiencing 5 or more bit errors to be a lost frame. If a frame experiences from 1 to 4 bits in error, it is considered to be a frame in ERROR. When all bits arrive correctly, we consider this a successful arrival of the frame (NOERROR). We can compute these probabilities for a frame of length L as follows.

$$\mathrm{P}_{\mathrm{NOERROR}} = (1 - \mathrm{BER})^{L}$$

$$\mathrm{P}_{\mathrm{ERROR}} = \sum_{k=1}^{4} \binom{L}{k} BER^{k} \, (1 - \mathrm{BER})^{(L-k)}$$

$$P_{LOSS} = 1 - P_{NOERROR} - P_{ERROR}$$

Note that we do not expect you to use the above stated formulas to generate the NOERROR, ERROR or LOSS event. We expect you to run L iterations. In each iteration, you generate 0 with probability BER, and 1 with probability 1-BER. You then count the number of zeroes. The number of zeroes represents the number of bits in error and thus corresponds to one of the three events.
Functionally speaking, the channel will take the time t_c, SN and the length of the frame/ACK as inputs and will either return nothing (NIL) if the frame/ACK is lost or return the time (after adding the delay τ), a flag representing whether the frame/ACK encountered an error or not, as well as the SN (which is unchanged).

3) **ABP Receiver**: The receiver maintains a NEXT_EXPECTED_FRAME counter, initialized to 0. The current time t_{cs} is also initialized to 0. The receiver acts on the return values of the forward channel. It updates its current time t_{cs} based on the channel return value. If the channel returns a frame with no errors, it checks the frame's SN and if it is equal to the NEXT_EXPECTED_FRAME, it counts it as a new and successfully delivered packet, which it sends to Layer 3 (in our case, it just increments a counter). This successful delivery is accompanied by increasing NEXT_EXPECTED_FRAME by 1 (mod 2). In all cases, it sends an ACK to the reverse channel of fixed length H, with the sequence number RN set to NEXT_EXPECTED_FRAME.
Note that the return values from the reverse channel will be registered in the ES as described earlier.

The forward and the reverse channel are functionally equivalent and hence should be implemented by one function definition. Also, it is important to make sure that correct tracking of time is carried out at both the sender and receiver implementation. As said already, the forward channel, receiver and the reverse channel should be wrapped into one functional block, to be used by the sender.

Details of the GBN Simulator

1) **GBN Sender**: The sender has a buffer that can hold N packets. Let us describe the buffer in some detail.

Buffer: The buffer has N spaces to hold up to N packets. Let **M**[i] represent the packet in the ith buffer space, for i = 1, 2, …, N. The index i increases from *left to right*. The buffer maintains the order with which the packets were generated in layer 3 in the sense that **M**[1] is older than **M**[2], **M**[2] is older than **M**[3], and so on. Let SN[i] and L[i] respectively represent the frame sequence number and the length of packet **M**[i]. When we have to slide the window, we are going to "push" the packets from the right to the left and hence P = SN[1] will always be the sequence number of the oldest packet in the buffer. Figure 5 depicts the buffer for N=4.

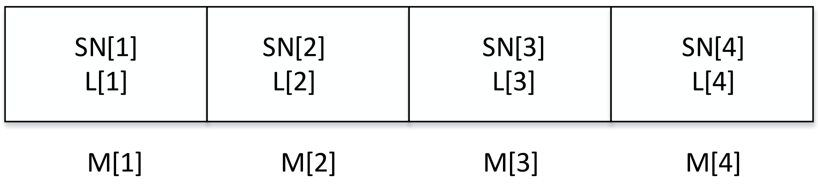


Figure 5 A buffer for N = 4

Initialization: At the beginning, the buffer is empty. The simulation begins by the buffer being filled with N packets, with each packet having a length L = H + *l*, where H is the fixed length of the frame header and *l* is the packet length assumed to be equal for all packets. The packets are assigned sequence numbers beginning with SN = 0 up to SN = N-1 in a rotating fashion, i.e., modulo N+1. So, we have

$$SN[1] = 0,\ SN[2] = SN[1] + 1, \text{ and so on.}$$
$$L[i] = H + l \text{ for } i = 1, 2, \ldots, N$$
$$P = SN[1] = 0$$

The sender also chooses a value for the time-out duration Δ, which is kept constant throughout the simulation. The *current time* t_c is initialized to 0.

At $t_c = 0$, the sender begins to transmit sequentially the N packets ($i = 1, 2, \ldots$ N) in the buffer, beginning with i=1. It also records the time T[i] at which the ith packet in the buffer is sent completely into the channel. Note, T[1] = t_c + L[1]/C, T[2] = T[1] + L[2]/C, and so on. Also, the sender registers a TIME-OUT event in the ES at time T[1] + Δ. Sending each packet involves calling a function that implements *together* 1) the *forward channel* (from the sender to the receiver), 2) the *receiver* as well as 3) the *reverse channel* (from the receiver to the sender). Like before, let this function be called *SEND()*. As explained already, this function returns as a value an event that has to be registered in the ES (See Fig. 4). Also, recall that if a frame (on the forward channel) or the ACK (on the reverse channel) gets lost, the function returns a NIL event which will not be registered in the ES. In other words, the return value of *SEND()* corresponds to a possible (i.e., not always) ACK event, which has a type: error or no error, and has a sequence-number RN. Note that, even though a *batch* of N packets are scheduled to be transmitted, the sender reads the ES after each packet in the batch is sent to receiver to find out if an event (an ACK or a TIME-OUT) has occurred during this time. i.e.,

After sending the ith packet:

- If no event (neither the receipt of an ACK or a time-out) occurred during the transmission of the ith packet, the sender continues and sends immediately the next packet, i.e., the (i+1)th packet, and so on.
- If one or more events had occurred during the transmission of the ith packet, the sender has to respond to the first of these events, called the *next event*, as follows:
    - If the *next event* is a TIME-OUT event (meaning that a time-out event had occurred between the start and the end of the transmission of the ith packet), all packets in the buffer are scheduled to be retransmitted immediately. The values of T[i]'s are also updated to reflect the new times when they were sent to the channel. A new TIME-OUT event has to be registered in the ES at a time T[1] + Δ. As explained before, sender has to read the ES after each packet in the current batch is sent to the receiver.

- If the *next event* is an ACK without error and RN is one of the expected sequence numbers (P+1, P+2, ..., P+N) modulo N+1, the sender knows one or more packets have been received correctly. RN = P+1 means that the first packet in the buffer was received successfully, RN = P+ 2 means that both the first and the second packets in the buffer were received successfully, and so on. The sender *slides* its window by (RN – P) where the subtraction has to be carried out modulo N+1. You can think of it as shifting the buffer by (RN – P) steps to the left (since we assume that the buffer index *i* increases from left to right) and filling the (RN - P) emptied buffer spaces with the required number of new packets with new random lengths, with their sequence numbers allocated sequentially. This *shifting and filling* results in an updated value of P = RN, as well as updated values of SN[i]'s, L[i]'s and T[i]'s. Note that since T[1] is updated, a new TIME-OUT event has to be created at T[1] + Δ. For example, for N = 4, if the sender receives an ACK with RN = 2 when P = SN[1]= 0, the shifting and filling will update the buffer as shown in Figure 6. After the shifting and filling is carried out, the sender schedules to *send* all the new (RN - P) packets. Note that, this can happen only after the sender is done with sending the old packets in the current buffer. Hence, if the window *shifting and filling* occurs in the middle of the batch transmission due to an ACK arriving in the middle, the sending of new packets can happen only after all the old packets are sent.
- If the next event is an ACK with error or its RN is **not** the one that is expected, our GBN sender ignores it. In other words, it continues transmitting the next packet of the current window as if no event had occurred.

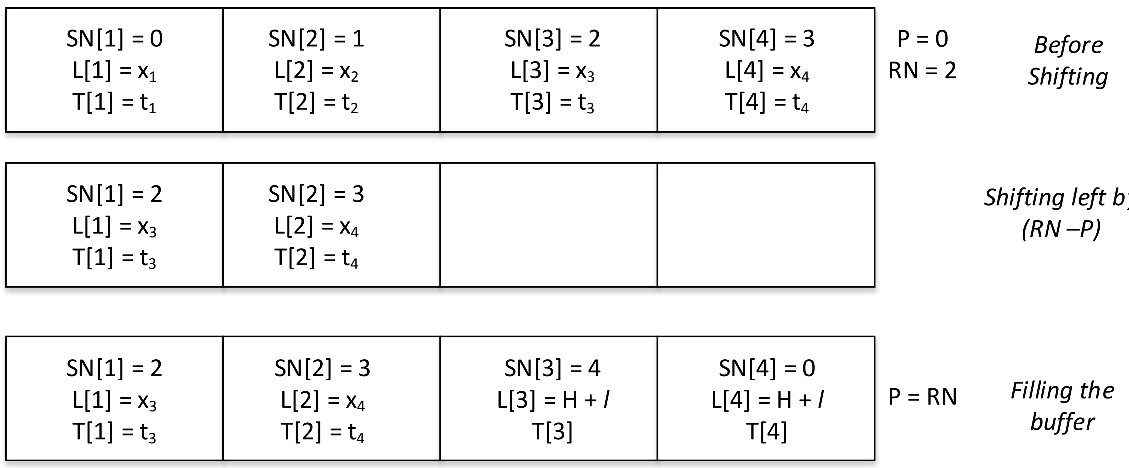


Figure 6 Shifting and filling as an implementation of sliding the window

- If the current sent packet is the last (i.e., the *N*th) packet in the current window, the sender dequeues the next earliest event from ES and processes it according to the above mentioned behavior.

Note. While the sender is sending packet *i*, starting at t_c, it cannot respond to events that occur during that transmission (i.e., between t_c and t_c + L[i]/C). An event that occurred during this interval has to be assumed to have occurred at t_c + L[i]/C even though it occurred before this time.

Note. Whenever a retransmission (as a response to a TIME-OUT event) or window-sliding occurs, the TIME-OUT event will be updated to a new time, corresponding to the time T[1] + Δ where T[1] is the time when the oldest packet in the buffer was sent to the channel. There can be only one time-out in the ES, and hence any outstanding time-outs in ES have to be purged before a new time-out is registered.

2) The channel behaviour is exactly the same as the one defined for ABP.

3) **GBN receiver**: The GBN receiver is very similar to that of an ABP receiver, except in terms of the set of valid sequence numbers. Hence, you can refer to the earlier discussion on ABP for channel and receiver behaviour.

4. Experiment:

A simulation experiment will be comprised of the following parameters.

1) Sender-side parameters: H, *l*, Δ
2) Channel parameters: C, τ, BER
3) Experiment duration in terms of *number of successfully delivered packets* to be simulated

These parameters will affect the performance of an ARQ protocol. In this experiment, you will need to investigate how a change in one or more of these parameters affects the throughput of an ARQ protocol. We will assume that the size of the frame header (which is also the size of an ACK) is fixed to H=54 bytes and the packet length is fixed to *l*=1500 bytes (i.e., a data frame has a constant size of H + l).

Question 1 (*ABP*)

Implement the ABP sender, channel and receiver as described above using the first option (namely the sender does nothing when it receives an ACK in error or a correct ACK with an RN not equal to NEXT_EXPECTED_ACK). Call this simulator *ABP*. Also, bundle the forward channel, receiver and the reverse channel components into one procedure. Also, implement an event scheduler, with a function for registering an event and a function for dequeueing the earliest event as well as removing timeout events from the queue when a new timeout is added. With the necessary components in place, integrate them to implement the simulator. Accompany your code with build scripts, and brief documentation explaining the key components of your implementation.

i) Consider the scenario where BER = 0. Use your simulator to compute the throughput (bits/sec) as a function of Δ for C = 5Mb/s, and two values of 2τ (10ms and 500ms). Note that while computing the throughput, you should not count the header bits. Simulate at least 10,000 successfully delivered packets and use the same methodology as in Lab 1 to obtain your results (i.e., you simulate for 5,000 successfully delivered packets, record the throughput, and then simulate for 5,000 more and see if the results are comparable). Take the values of Δ from the set {2.5 τ, 5τ, 7.5τ, 10τ, 12.5τ}.

ii) Now, repeat the set of experiments in (i) with BER = 1.0e-5 and BER = 1.0e-4.

Question 2 *(ABP_NAK)*

Note that the ABP sender in Question 1 ignores an ACK that arrives in error or with RN not equal to NEXT_EXPECTED_ACK. The sender can however take these events as negative acknowledgements (NAK) and act on them by resending the same packet as soon as such events occur. Implement such an alternative version of ABP sender. Call this simulator *ABP_NAK*. Use your simulator to obtain the same set of results as in Question 1 and compare the experimental findings with that of Question 1. Discuss the findings. Refer to the submission guidelines in the Appendix for the set of plots that you are required to submit for Questions 1 and 2.

Question 3 *(GBN)*

Implement the GBN sender as described above. Change your ABP receiver code to account for the larger set of sequence numbers used in GBN. Reuse the channel function that you implemented for Question 1. Integrate your components to implement a GBN simulator. Call this simulator *GBN*. You will study similar scenarios for GBN as you considered for ABP.

i) Take N = 4 and BER = 0. Use your simulator to compute the throughput (bits/sec) as a function of Δ for C = 5Mb/s, and two values of 2τ (10ms and 500ms). Compare your results with that of Question 1.i by putting the results for *ABP* on the same graph.

ii) Repeat Question 1.ii for GBN. Plot your results together with that of results obtained in Question 1.ii for *ABP*. Discuss the results.

What To Turn In:

Complete source codes:

- Simulator code for Question 1.
- Simulator code for Question 2.
- Simulator code for Question 3.

Result files (CSV Files) (Refer to Appendix for the format of the CSV files):

- *ABP.csv*
- *ABP_NAK.csv*
- *GBN.csv*

Simulator Run Scripts

- *run_ABP*
- *run_ABP_NAK*
- *run_GBN*
- **Important note**: Your code has to compile and run successfully on *ecelinux.uwaterloo.ca*.

A report file (pdf) containing at least

- A brief explanation of each of your three simulators
- Answers to questions along with the relevant plots

Put all required files/folders into an archive file and upload to LEARN dropbox (no need for a hardcopy submission).

Refer to the Appendix for the submission guidelines. Read them carefully.

Appendix: Submission Guideline

A1. Summary of result data

The following table summarizes the throughput results that you need to generate for each of the three questions.

Δ/τ	**2τ = 10ms**			**2τ = 500ms**		
	BER=0.0	BER=1e-5	BER=1e-4	BER=0.0	BER=1e-5	BER=1e-4
2.5	a1	a2	a3	a4	a5	a6
5	b1	b2	b3	b4	b5	b6
7.5	c1	c2	c3	c4	c5	c6
10	d1	d2	d3	d4	d5	d6
12.5	e1	e2	e3	e4	e5	e6

❑ Hence your report should include the above table for each of the three simulators.

❑ **You also need to attach the corresponding CSV file for each of the three simulators.** The files should be named *ABP.csv* for ABP (Question 1), *ABP_NAK.csv* for ABP_NAK (Question 2), and *GBN.csv* for GBN (Question 3).

The fields of each CSV file should be exactly as shown below. Express throughputs in *bits/sec*.

```
a1,a2,a3,a4,a5,a6
b1,b2,b3,b4,b5,b6
c1,c2,c3,c4,c5,c6
d1,d2,d3,d4,d5,d6
e1,e2,e3,e4,e5,e6
```

A2. Simulator Commands

We should be able to compile and run your simulator(s) by using ***one command***. For running ABP, ABP_NAK and GBN, you should include script files *run_ABP, run_ABP_NAK* and *run_GBN*, respectively.

For example, we should be able to run ABP simulator (Question 1) simply by running the following command ($ is not the part of the command):

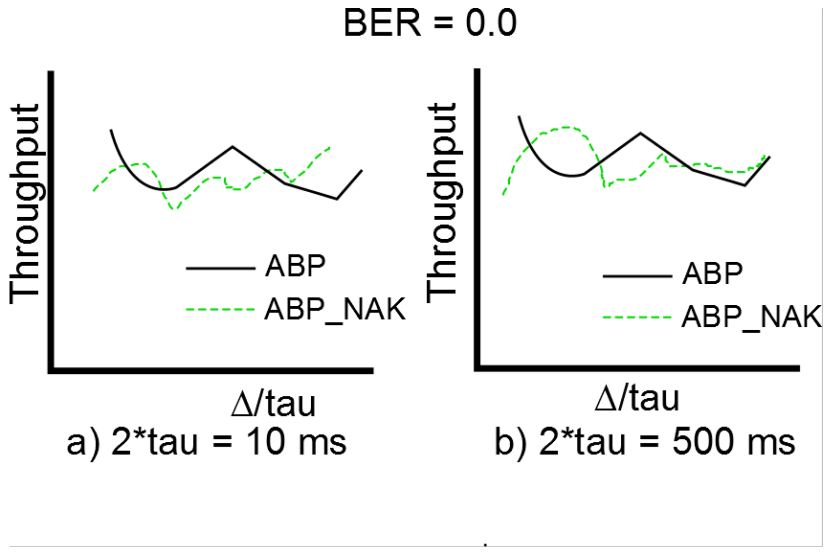
```
$ run_ABP
```

The output of the run command should be the corresponding result file in CSV format (whose fields are explained before). For example, running ABP with the above mentioned command should generate *ABP.csv*.

You might need to learn about **gnu make** (or other build tools) and some basic **bash shell scripts** to be able to fulfill this requirement.

A3. Summary of required plots (for the report)

❑ Include plots for Q1 and Q2 together (as shown) and discuss the findings in your report.



Include similar plots for BER = 1.0e-5 and 1.0e-4.

❑ Similarly, include plots for Q1 and Q3 together and discuss the findings in your report.

<Cover Page>

ECE358: Computer Networks

Winter 2014

Project 2: Data Link Layers and ARQ Protocols

Date of submission:

Submitted by:

Student ID:

Student name <Last name, First name>

Waterloo Email address

Marks received: <Leave this blank>

Marked by: <Leave this blank>