LAB MANUAL

ECE 358 - Project 2

Data Link Layer and ARQ Protocols

Deadline: Monday, July 9th 2012

Table of Content

Objective

Equipment

Overview

Background Material

(1) Alternative Bit Protocol (ABP)

(2) Go Back N

(3) Selective Repeat Protocol (SRP)

(4) Important Parameters

(5) Pseudocode

(6) Flowchart

Instructions

(1) How to use library function lab2linux.o

(2) Graph Plotting

Questions

Final Report

Supplementary Material

(1) Library Function (lab2linux.o)

(2) main.c

(3) lab2.h

OBJECTIVE

To design a simulator, and use it to understand the behaviour of three ARQ (Automatic Repeat Request) protocols: Alternative Bit Protocol (ABP), Go Back N and Selective Repeat Protocol (SRP)

After this experiment, the students should understand:

i. How a data link layer and different 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.

EQUIPMENT

A computer with C compiler and library functions provided by the instructor.

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, and most local area networks are treated as single links from the point of view of the data link layer (called LLC (Logical Link Control) in that context, performing on top of the MAC (Multiple-Access Control) sub-layer). On the sending side the data link layer encapsulates protocol data units (PDUs), also known as packets, coming from the network layer 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. 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.

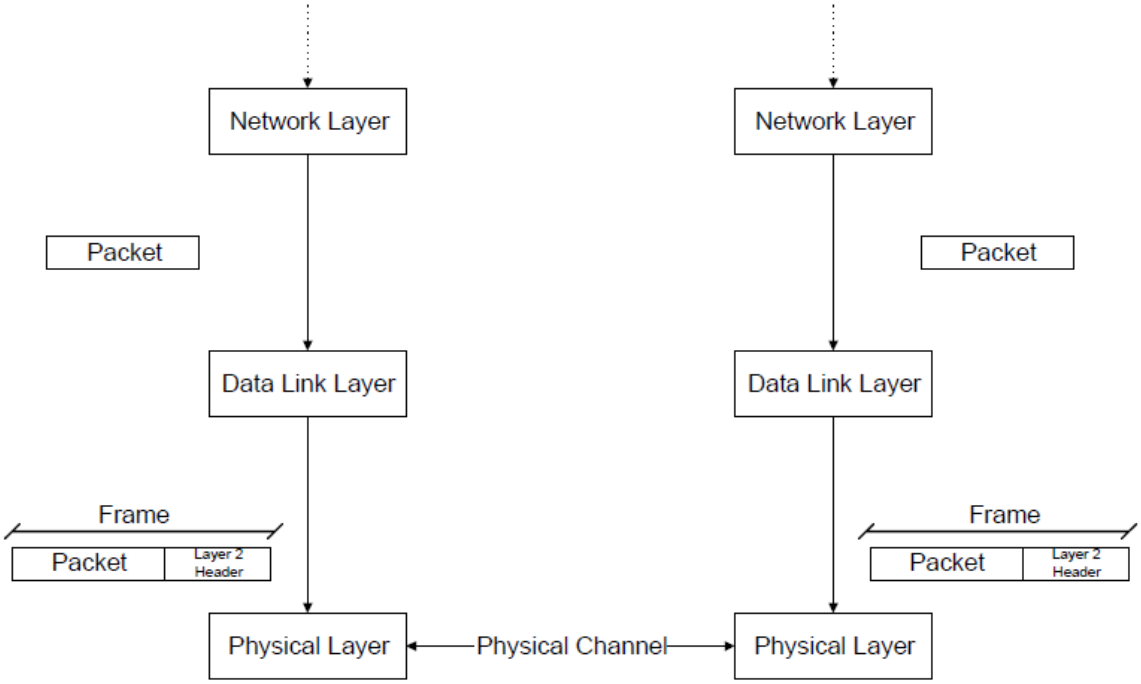

Figure 1 Relationship between packets and frames

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* to the frame and the receiver will detect errors and request for retransmission if errors occur. Although ARQ involves extra overhead for retransmission requests and retransmitting frames in error, 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 data link layers, but also to transport layers 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.

BACKGROUND MATERIALS

In this experiment, we will investigate three different ARQ protocols, namely, *Alternating Bit Protocol (ABP)*, *Go Back N (GBN)*, and *Selective Repeat Protocol (SRP)*. 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 the side receiving it and sending acknowledgements (ACK).

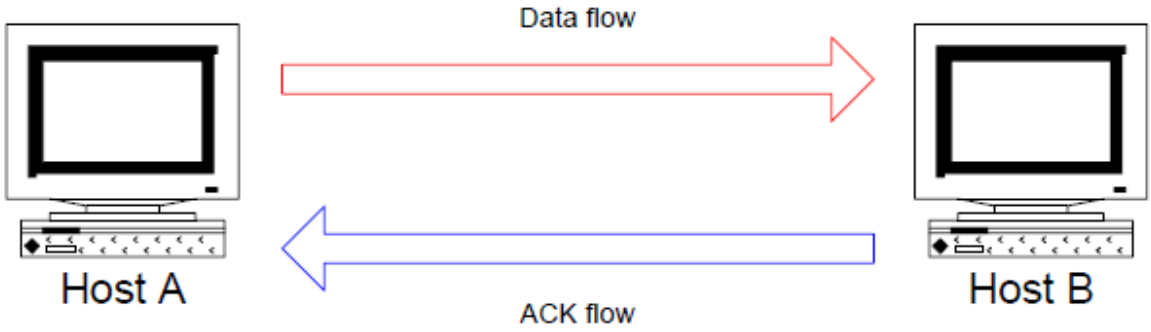

Figure 2

I. Alternative Bit Protocol (ABP)

In ABP, the sender 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 till 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 receiver acknowledges every correct frame it receives 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 corresponding to the sequence number of the received data frame coming from A. The receiver keeps counter EXT_EXPECTED_FRAME and sends the packet received in a correct frame to the upper layer ONLY if the sequence number is equal to the value of that counter. In that case, it increments the counter by one (modulo 1). The sender will wait for the receiver's ACK with the right ACK number before emptying its buffer of the current packet and sending a new packet in a new numbered frame (assuming that the layer 3 of A has one to send). If the sender does not receive the acknowledgement before the frame's timeout, either because the sent frame or the ACK is not correctly received, the sender will retransmit the same packet in a new frame carrying the same sequence number as the original frame.

II. Go Back N

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. The sender has a pointer P that points to the frame sequence number of the lowest numbered packet not yet acknowledged or to NIL if there is not such a 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. P can only be updated (i.e., we can only slide the window) when the frame numbered P has been acknowledged. We will assume that the receiver will acknowledge every correctly received frame by sending an ACK with a number corresponding to the last in order frame correctly received. (Note that in the real protocol, the receiver will acknowledge every correctly received frame by requesting the next expected frame). Hence if B is expecting frame 8 (i.e., it has receives all frames up to 7 correctly) and receives correctly frame 9, it acknowledges it by sending an ACK with number 7 (which means that it has received correctly all frames up to 7 (included)). In Go Back N, the receiver would not keep the packet contained in frame 9 but would discard it. If a frame is received in order, its corresponding packet will be forwarded to the upper layer and if not, it will be discarded. On the other hand, if the sender does not receive the ACK of a frame *k* before a timeout, it will resend

the packet that was in frame *k* and *all the packets sent after frame k*. These retransmissions are required because the receiver will discard all out-of-order frames, so the sender cannot just send the lost frame only.

III. Selective Repeat Protocol

SRP is a smarter protocol in which only the missing/corrupted packets are required to be retransmitted. SRP works almost like GBN. The sender has a pointer P that points to the frame sequence number of the lowest numbered packet not yet acknowledged or to NIL if there is not such a packet. The sender can send at most N packets in frames numbered P,…, P+N-1 (modulo 2N+1) (see later) before receiving a correct acknowledgement for frame numbered P. P can only be updated (i.e. we can only slide the window) when the frame numbered P has been acknowledged. If a packet is lost, the sender only retransmits the lost packet because the receiver has a buffer of size N and keeps correctly delivered packets even when out of order as long as their number is less than the number of the "next in order expected frame" + N – 1, modulo 2N+1 (see later). Hence in SRP, both the sender and the receiver have a buffer able to contain N packets. The sender uses its buffer to store the packets not yet acknowledged and the receiver uses its buffer to store out-of-order packets. The receiver delivers packets to the upper layer in order. The window size N has the same meaning as GBN, but the numbering of frames is different. Instead of modulo (N+1), a numbering modulo 2N+1 should be used in order to avoid ambiguity. The receiver acknowledges each frame individually. Hence if B is expecting frame 8 (i.e., it has received all frames up to 7 correctly) and receives correctly frame 9, it acknowledges it by sending an ACK with number 9. When the sender receives an out-of order ACK, it can discard the corresponding packet but does not slide the window if the acknowledged packet was not the first of the window (modulo 2N+1).

* Note that when the window size equals 1, both GBN and SRP will reduce to ABP.

IV. Important Parameters

The parameters that will affect the performance of ARQ protocols are the *channel capacity* C, the *propagation delay* τ, the *length of a data frame* L (assumed constant, but it is not true in general), and the *length of a ACK frame* A (assumed constant, but it is not true in general).

Since CRC is not part of the experiment, we will not work with bit error rates but instead, we will use *frame error rates* (FER). Let us define p(x) the FER as the probability that a frame of size x is corrupted. Note that p(x) is **roughly** proportional to x, i.e., p(L) ≈ L*BER. In reality the function is NOT linear but we will make this assumption for easy computation of FER.

In this experiment, you will need to investigate how a change in C and/or τ affects the throughput of the different ARQ protocols under different FER. We will fix L=1500 bytes and A=54 bytes. Figure 3 shows the model you are going to simulate:

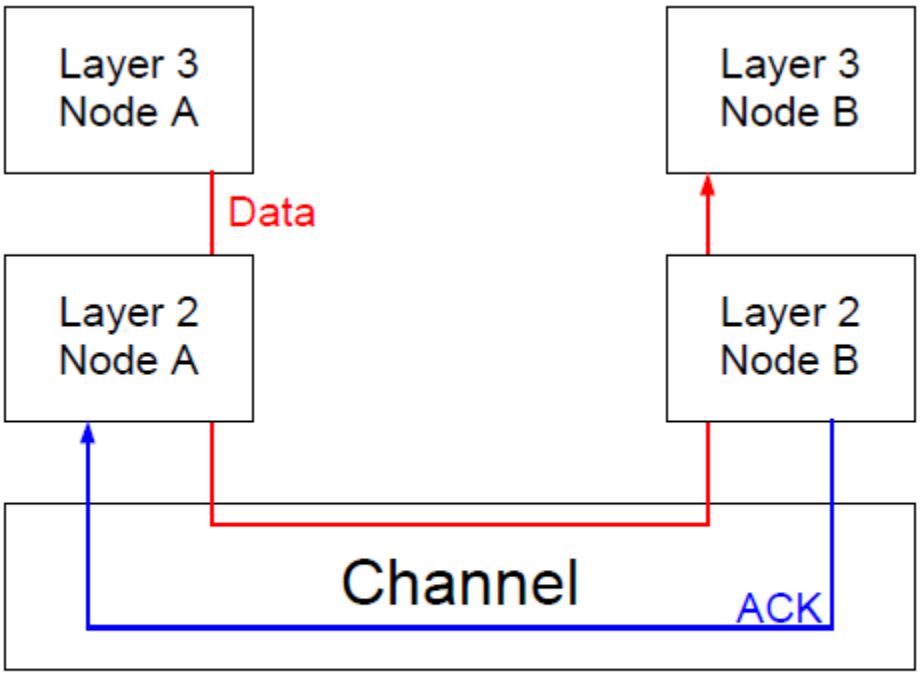

Figure 3: Model for simulation

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.

V. Pseudocode

A sample `main()` function is included for your reference. You can directly use it if you want to, and add the `Sender()` and `Receiver()` functions before the main function.

This main function is available in the web page.

```
#include <stdio.h>
#include <stdlib.h>
#include "lab2.h"
struct Queue *Queue_Head = NULL;
double C;
double A;
double L;
double Prop_Delay;
double Time_Out;
double FER;
int Window_Size;
int N; /* Total number of packets */
void Sender(Event Current_Event) {
/* You sender code here */
}
void Receiver(Event Current_Event) {
/* Your receiver code here */
}
int main()
{
Event Current_Event;
/*******************************************/
/* Remember to change the following variables */
N = 25;
C = 1000000; /* bps */
L = 1500*8; /* bits */
A = 54*8; /* bits */
Prop_delay = .050; /* seconds */
Window_Size = 5;
FER = 0.01;
Time_Out = 0.112432;
/*******************************************/
Initialization();#include <stdio.h>
#include <stdlib.h>
#include "lab2.h"
struct Queue *Queue_Head = NULL;
double C;
double A;
double L;
double Prop_Delay;
double Time_Out;
double FER;
int Window_Size;
int N; /* Total number of packets */
void Sender(Event Current_Event) {
/* You sender code here */
}
void Receiver(Event Current_Event) {
/* Your receiver code here */
```

```
}
int main()
{
Event Current_Event;
/*****************************************/
/* Remember to change the following variables */
N = 25;
C = 1000000; /* bps */
L = 1500*8; /* bits */
A = 54*8; /* bits */
Prop_delay = 50; /* seconds */
Window_Size = 5;
FER = 0.01;
Time_Out = 10;
/*****************************************/
Initialization();
while (Queue_Head != NULL)
{
Dequeue(&Current_Event);
if ( (Current_Event.Type == RECEIVE_ACK)
|| (Current_Event.Type == START_SEND)
|| (Current_Event.Type == TIMEOUT))
{
Print(Current_Event);
Sender(Current_Event);
}
else if (Current_Event.Type == RECEIVE_FRAME)
{
Print(Current_Event);
Receiver(Current_Event);
}
}
return 0;
}
```

VI. Flowchart

We will use the DES (Discrete Event Simulation) framework introduced in experiment 1. A queue will be used to store the events in sequence and the function `Dequeue()` will be used to return the event at the head of the queue (the next earliest happening event). The structure of an event has been defined for you and you are going to use it throughout the experiment:

```
typedef struct
{
 float Time; /* The time at which the event happens */
 int Type; /* What event it is. See explanations below */
 int Seq_Num; /* The frame number sender and receiver will use */
 int Pkt_Num; /* The packet number */
 int Error; /* 0: no error, 1:has error */
}Event;
```

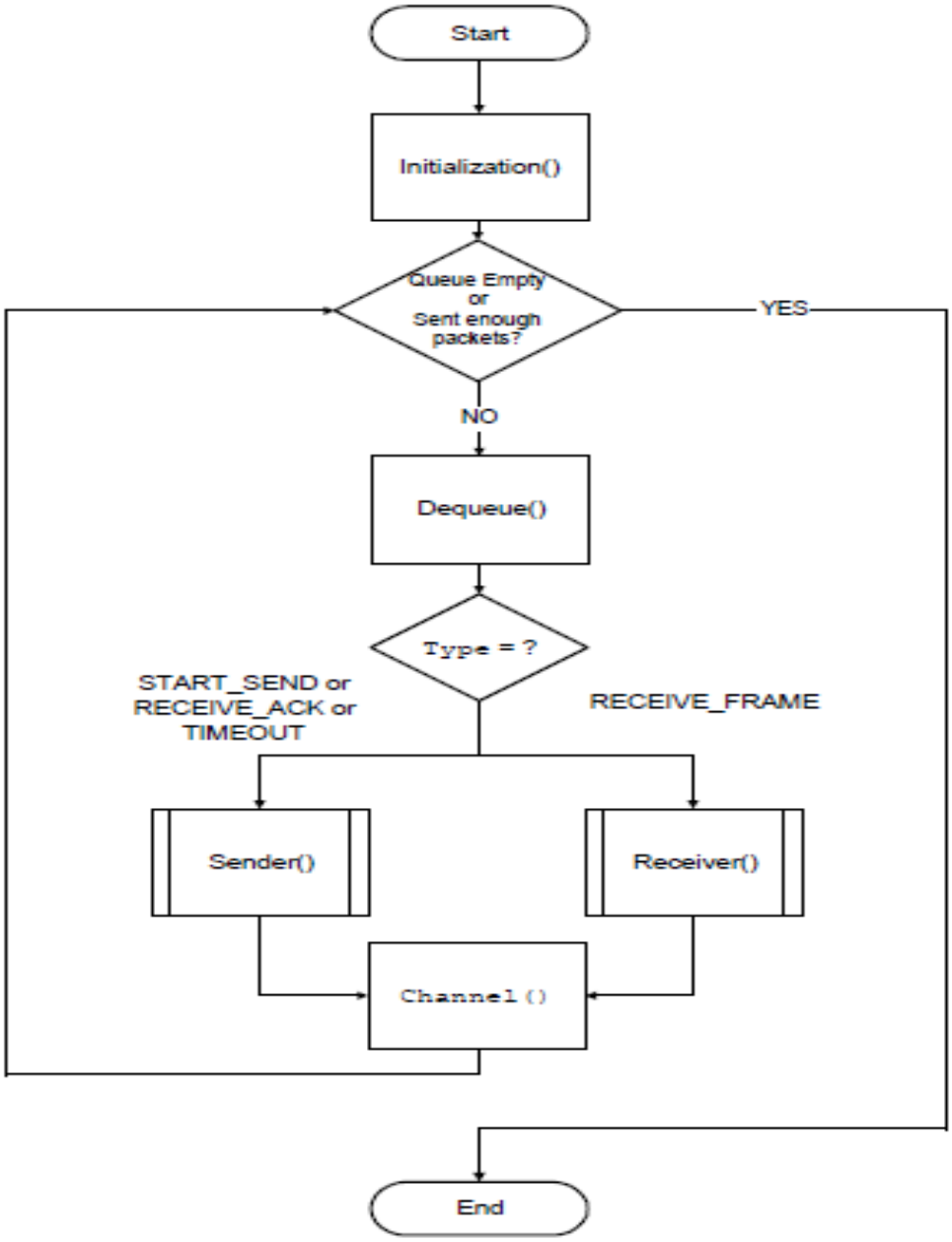

Figure 4: Flowchart of the simulator

The fields in an `Event` are interpreted as follow: `Time` is the time at which this event happens; `Type` is the type of this event in symbolic constants, which can be START_SEND (sender starts sending a data frame), RECEIVE_ACK (ACK is received at sender) or TIMEOUT (a frame is timed out) for the sender to take action; or RECEIVE_FRAME (a frame is received at the receiver) for the receiver to take action. The `Seq_Num` and the `Pkt_Num` are the frame

sequence number and the number of the packet in the frame involved in this event respectively. The field `Error` indicates whether a frame (data or ACK) is erroneous or not.

The functions `Initialization()` and `Dequeue()` are included in the library. `Initialization()` will initialize the system and make it ready for simulation. It will also generate the first batch of send event according to the parameters of the system. The usage of the function is straightforward:

```
Initialization();
```

`Dequeue()` will give you the event next to happen. Your job is to write the `Sender()` and `Receiver()` functions, according to the ARQ protocol you are going to implement. The usage of `Dequeue` is as follow:

```
Dequeue(&Event);
```

The function will return the *next* event that will happen. You can then pass the `Event` to either the `Sender()` function or the `Receiver()` function depending on what `Type` the `Event` is. Your `Sender()` and `Receiver()` functions should have the following prototype (i.e., format): (see example in Figure 5)

```
Sender(Event Current_Event);
```

and

```
Receiver(Event Current_Event);
```

Another function you will use in the library is the `Channel()` function. `Channel()` simulate the loss(es)/corruption(s) happening in a physical channel. The usage of `Channel()` is as follow:

```
Channel(int Type, int Seq_Num, int Pkt_Num, double Current_Time);
```

You should pass the type of your action, which is either SEND_FRAME or SEND_ACK; the Sequence Number of the data frame/ACK frame you will send; the Packet Number of the data packet you will send or 0 if it is an ACK; and the current time to the channel. The `Channel()` function will enqueue your action and create events for simulation. So by calling `Dequeue()` repeatedly, and by reacting according to the event type, you can simulate the performance of ARQ protocols.

When a data frame is correctly received at the receiver side, and is ready to pass to layer 3, you should call:

```
Deliver(Event Current_Event, double current_time);
```

to deliver the in-order packets. For SRP receiver, you will also need a buffer to store out-of-order events and deliver them to layer 3 when ready.

Two files, namely layer2.txt and layer3.txt will be created by the function `Print()`. These text files should be used for debugging, and to check the validity of your protocol.

In your program, you should have the following variables set before you start calling `Initialization()`: C (channel capacity), τ (one-way propagation delay), L (length of data frame), A (length of ACK frame) and FER (Frame error ratio of data frame. The program will compute the error ratio of ACK frame accordingly). You also need to set the values used by the protocol: Window_Size (window size) and Time_Out (timeout). If the processing delay is negligible, that is, a receiver will send an ACK as soon as it receives a correct data frame, then the value of timeout should be a value slightly greater than ($L/C + A/C + 2\tau$).

INSTRUCTIONS

I. Library File (`lab2linux.o`)

In order to begin the experiment, you should download the library file `lab2linux.o` from the web site. `lab2linux.o` was tested on Linux machines. All the structures and predefined values, as well as the procedures you are going to need for the experiment are included in this file.

How to use the library file?

In this experiment, you will be asked to build the senders and receivers for different ARQ protocols. The channel and other structure definitions are given to you (embedded in the library file) and explanations follow. When you compile your program in Linux, use the command:

```
gcc lab2linux.o your_file.c
```

You may also add other options of gcc if you need them. Figure 4 contains the flowchart of the simulator. Make sure you place the header file (lab2.h), object file (`lab2linux.o`) and `your_file.c` in the same folder.

II. **Graph Plotting**

Plot graphs for relevant questions. Comment on graphs, like what is your observation.

**Additional instructions are given with each question.*

QUESTIONS

Question 1: Write the procedures `Sender()`, `Receiver()` for ABP. Explain what you do. Giving the code is not enough.

For the following settings: L and A are fixed at 1500 bytes and 54 bytes respectively, and C is either 1Mb/s or 10Mb/s, τ is either 50ms or 500ms, Time_Out is set to ($L/C + A/C + 2\tau$) correspondingly and FER is 1% or 2%. Send 10,000 packets. Show your experimental findings, i.e., the time you need to send 10,000 packets successfully for each case, the efficiency and the thoughput (average number of packets sent to the layer 3 of B per second). How do your results compare to the formulas derived in class?

Question 2: Write your own `Sender()`, `Receiver()` for GBN. Explain what you do. Giving the code is not enough. The functions `Sender_GBN()` and `Receiver_GBN()` are included in the library and are ready to work. You can use them to test the correctness of your code. You can start by using `Sender_GBN()` to match with your own `Receiver()`, and the other way around. After your own `Sender()`/ `Receiver()` are working perfectly with the library functions, you can try to combine them and see if they can work with each other by comparing the result of using only `Sender_GBN()` and `Receiver_GBN()`. In order to make sure that the results that you will give us are from your own simulator and not from ours, we have set the frame size in our simulator to 500 bits and the ACK size to 20 bits.

Question 3: We believe that writing your own Sender() and Receiver() for SRP is too complex and time consuming. Hence we are providing you the 2 functions so that you can compare the performances of SRP against those of GBN and ABP for different sets of parameters. The functions `Sender_SRP()` and `Receiver_SRP()` are included in the library and are ready to work. Use the functions you obtained in question 2 and the SRP simulator that is provided to you to compare GBN and SRP by plotting the following curves: (Plot both GBN and SRP on the same graph) Fix: L=1500 bytes, A=54 bytes, C=1Mb/s, τ=50ms. Plot throughput (average

number of packets sent to the layer 3 of B per second) vs FER and efficiency (on 2 different figures) vs. FER, where FER ranges from 10% to 90%, step size 10% and also at FER=5%. Stop when 500,000 packets are successfully sent. Repeat for Window_Size equal 5 and 15. For each case, comment the results and compare with the formulas derived or given in class.

Question 4: Repeat Question 3 by fixing L=1500 bytes, A=54 bytes, FER=5%, C=1Mb/s. Plot throughput vs. τ and efficiency vs. τ, where τ ranges from 20ms to 500ms, step size 40ms. Stop when 500,000 packets are successfully sent. Repeat for Window_Size equal 5 and 15. For each case, comment the results.

Question 5: Repeat Question 3 by fixing L=1500 bytes, A=54 bytes, τ =50ms, FER=5%. Plot throughput vs. C and efficiency vs. C, where C ranges from 0.5 Mb/s to 5 Mb/s, step size 0.5 Mb/s. Stop when 500,000 packets are successfully sent. Repeat for Window_Size equal 5 and 15. For each case, comment the results.

FINAL REPORT

1. The source codes of `Sender()` and `Receiver()` for ABP with clear explanation of what you did.
2. Experimental results of question 1 and comments on the result.
3. The source codes of `Sender()` and `Receiver()` for GBN with clear explanation of what you did.
4. The graphs with comments for question 3 to question 5.

SUPPLEMENTARY MATERIAL

Enclosed is the library function lab2linux.o in C