Cpr E 489
Lab Experiment #4: Error Detection and Go-Back-N ARQ Protocol
(100 points in total)

Objective

To write code to implement the sender part of the Go-Back-N ARQ protocol.

Pre-Lab

Read over the provided source code that has already been implemented (`secondary.c`, `utilities.h/.c`, `ccitt16.h`, `introduceerror.h/.c`, and the skeleton in `primary.c`). Develop state machines and/or pseudo code to help implement your solution.

Lab Expectations

Work through the lab and let the TA know if you have any questions. **Demonstrate your program to the TA after you have completed it.** After the lab, write up a lab report. Be sure to

1) summarize what you learned in a few paragraphs. (**20 points**)
2) submit your well-commented code and demo to the TA. (**50 points**)
3) include your answer to the exercise. (**30 points**)
4) specify the effort levels of each group member. (totaling to 100%)

Your lab report is due at the beginning of the next lab. Be sure to submit your **well-commented code** to the TA with your lab report for grading. Submit your report as a PDF file, and your code as a `.c` file.

Login Information

You may login using your NetID or 489labuser, however, it would be a good idea to use your NetID as you will be able to have the code stored in your U-drive, and you can continue working on any University Linux machine that has `gcc`.

Problem Description

In this lab experiment, you are required to design and develop a sender function (`primary.c`) to implement the Go-Back-N ARQ protocol, with a Primary function (which will act as a client or sender) while the Secondary function (which will act as a server or receiver) is given to you. This Go-Back-N ARQ protocol is a revised version from the one we learned in class. One difference is that it uses both ACK and NAK packets; this is to simplify the implementation and avoid having to implement the timeout mechanism.

For CRC generation and error detection, you should use the provided object file (`ccitt16.o`). See the CRC notes below for more information.

You will be provided with the client (sender) and server (receiver) `.o` files. The sender establishes a TCP connection and hands control to your "primary" function, while the receiver accepts the TCP connection and hands control to your "secondary" function. The provided implementation already contains an example of two-way communication with comments explaining the different parameters necessary to send and receive data.

Primary Function (50 points)

- Send the alphabet, ABCDEFGHIJKLMNOPQRSTUVWXYZ, to the Secondary in a total of 13 packets (two characters per packet, NOTE: the total number of packets sent may be higher since the packets may be corrupted by `IntroduceError(.,.)` and thus need to be redelivered). (**10 points**)

- Use the "`buildPacket`" function that can be found in "`utilities.c`" to build a packet of the following format: (**10 points**)

Packet Type	Packet Number	Data	CRC
1 byte	1 byte	2 bytes	2 bytes

  - **Packet Type**
    - 1: Data Packet (sent from Primary to Secondary)
    - 2: Acknowledgement Packet (ACK) (sent from Secondary to Primary)
    - 3: Negative Acknowledgment Packet (NAK) (sent from Secondary to Primary)
  - **Packet Number**
    - Starts from 0 and increments sequentially to 12
  - **Data**
    - Two alphabet characters (sent from Primary to Secondary)
    - No data is sent from Secondary to Primary
  - **CRC**
    - CRC generated for this entire packet, including Packet Type, Packet Number, and Data fields (see the section on CRC)

- Display the packet. (`printPacket(.)`) (**10 points**)
- Apply the "`IntroduceError(.,.)`" routine to the entire packet. Pass the BER value to the program as an argument in the command line. (**10 points**)
- Implement the Go-Back-N ARQ protocol with a send window of size N = 3: (**10 points**)
  - The Primary sends the packet to the Secondary and keeps it in a buffer until an ACK is received for this packet. The Primary displays an indication of sending the packet, together with its sequence number.
  - When the Primary receives an ACK from the Secondary, it adjusts the send window and removes the packet from its buffer. It also displays an indication of which packet was positively acknowledged.
  - When the Primary receives a NAK from the Secondary, it adjusts the send window and retransmits all the packets in the send window. It also displays an indication of which packets were negatively acknowledged and were retransmitted.
  - Notes: the following two functions are found in "`utilities.c`":
    `shiftWindow(.,.,.)`: used to shift the send window frame.
    `shiftBuf(.,.,.)`: used to shift the packet buffer frame.

Secondary Function

(Note: The secondary function is completed and you don't need to make any modifications, but it's a good example to help you build the primary function.)

- Accept data packets from the Primary.
- Run the CRC check:
  - If the packet is received error free and in sequence, it displays the packet content and sequence number, then sends back an ACK.
  - If the packet is received error free but out of sequence, it does not display the packet content but displays the sequence number, then sends back an ACK.
  - If the packet is received in error, it does not display the packet content but displays the sequence number if possible, then sends back a NAK.
  - ACK and NAK packets are not corrupted.

Procedure

- Complete the Primary function as described above.
- Correctly transmit packets from the Primary (sender) to the Secondary (receiver) and observe the sequence of transmissions.
- Demonstrate your program to the TA.
- Submit your (well-commented) code to the TA with your lab report for grading.

Exercises (30 points)

1) Run your program five times with each of the following BER values: 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, and graph the average number of transmission attempts per data packet for each BER value. Summarize your observations.

Notes

- The localhost has the IP address of 127.0.0.1.
- In order to avoid contention and confusion, use a server port number that is equal to the rightmost five digits in your student number. For example, if your Student ID number is 123-45-6789, then the port number should be 56789. If this number is less than 1023, greater than 65535, or equal to any reserved port number (see the `/etc/services` file) you may use any random port number.

Compiling and Running "sender" and "receiver" Programs

Receiver

First, compile `secondary.c` with the `-c` option to generate an object file:

```
gcc -c secondary.c
gcc -c utilities.c
```

Then, compile both `.o` files together with `ccitt16.o` to create an executable:

```
gcc -o receiver receiver.o secondary.o utilities.o ccitt16.o
```

Finally, run `receiver`, and have it listened on some port number (change 50404 to another number according to the notes mentioned above).

```
./receiver 50404
```

Sender

First, compile `primary.c` with the `-c` option to generate an object file:

```
gcc -c primary.c
```

Then, compile both `.o` files together with `ccitt16.o` to create an executable:

```
gcc -o sender sender.o primary.o utilities.o ccitt16.o
```

Finally, run `sender`, and have it connected to the receiver program on a given IP and port number (that should match the receiver's). You are also required to pass the BER value that will be handed down to your primary function.

```
./sender 127.0.0.1 50404 0.001
```

CRC Generation and Checking

Compiling

An object file, `ccitt16.o`, is provided that will generate and check CRC for you. In order to use the `.o` file, you first need to include `ccitt16.h` with your file (e.g., `yourprogram.c`):

```
#include "ccitt16.h"
```

Now, compile your file with the `-c` option to generate a `.o` file:

```
gcc -c yourprogram.c
```

Finally, compile both `.o` files together to create an executable:

```
gcc -o yourprogram yourprogram.o ccitt16.o
```

Usage

The function provided by `ccitt16.o` has the following prototype:

```
short int calculate_CCITT16(unsigned char cData[], unsigned int iLen,
      unsigned int iAction);
```

`iAction` is defined as either `GENERATE_CRC` or `CHECK_CRC` in the `ccitt16.h` header file:

```
#define GENERATE_CRC    1
```

- Returns the checksum of `cData[]` with length `iLen` as a short `int`

```
#define CHECK_CRC       2
```

- Uses the last two bytes of `cData[]` as CRC check bits to check `cData[]`; returns either 0 or 1:
  - `#define CRC_CHECK_SUCCESSFUL  0`
  - `#define CRC_CHECK_FAILURE     1`