CS 39006: Networks Lab
Lab Test 1: Implementation of Sliding Window Flow Control
Date: 15th February, 2018

Time: 2 Hours

Submission Instructions:

The code needs to be written in either C or C++ programming language.

Include a **makefile** to compile your code. You need to submit the code (only C/C++ source) and Makefile in a single compressed (tar.gz) file. Rename the compressed file as `test_1_Roll.tar.gz`, where `Roll` is the roll number of the student. Submit the compressed file through Moodle by the submission deadline.

Write down your assumptions as a comment inside the code. No queries will be entertained during the exam.

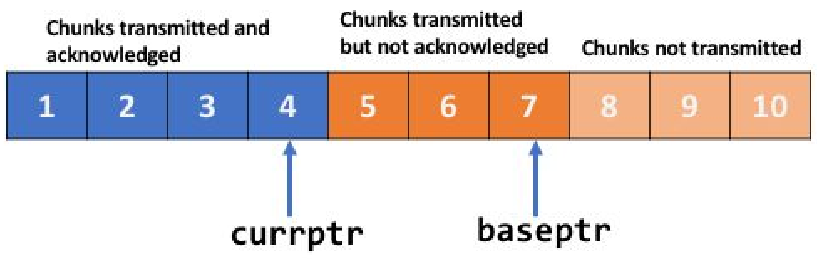
Problem Statement:

You have implemented a stop and wait ARQ based flow control and reliability mechanism on top of UDP in Assignment 3. During the lab test, you have to extend the implementation of flow control and reliability with Go Back N ARQ. The design requirements are as follows.

1. There would be a single file server and a single file client. The client sends a sufficiently large file (around 1 MB) to the server.
2. The transport layer protocol used for this file transfer is UDP.
3. Divide the file into chunks of 1 KB (last chunk can be less than 1 KB)
4. Inform the server about the **file name**, the **total file size** and the **number of chunks** to be sent
5. Add an application header to every chunk. Each header should contain following fields,
    a. **Sequence number** - 4 Bytes (per chunk sequence number)
    b. **Length of the chunk** - 4 Bytes

   This additional informations appended with the application header will help you to ensure the reliability.
6. Forward the chunks using a **Go Back N ARQ protocol** with `timeout=1` second, to ensure the reliability of data transfer at the application layer.
7. Go Back N works using following principles:

a. Use a sender window with window size 3. Every window can contain one chunk with the application layer header. This can be implemented using two pointers `baseptr` and `currptr`, where `baseptr` points to the sequence number upto which the chunks have been transmitted, and `currptr` points to to the sequence number upto which the chunks have been acknowledged. So, `baseptr - currptr <= 3` always.


Chunks transmitted and acknowledged
Chunks transmitted but not acknowledged
Chunks not transmitted
1
2
3
4
5
6
7
8
9
10
currptr
baseptr

b. The sender can send at most N number of packets without waiting for an ACK, where N is the size of the window (here 3).
c. Once the sender receives an ACK, the sender progress the window further to send more packets.
d. The ACK contains a cumulative acknowledgement number, where acknowledgement number N indicates that all chunks upto the chunk N has been acknowledged, and the receiver is expecting chunk N+1.
e. After sending all the chunks up to the window size, the sender starts a timer with `timeout=1` second. If the ACK is received before the timer expires, the sender sends more packets and restart the timer. If the timer expires before the ACK is received, all the chunks inside the current window are retransmitted.

Additional Notes:

1. You can use a window based timeout, where the timer is started when the window is full (`baseptr - currptr = 3`), and stopped when an ACK corresponds to at least one chunk is received (do not wait for all the transmitted chunks). You can implement a timer using `SIGALRM` signal. A sample code is given at the end. However, if you wish, you can implement timeout based on your own implementation.
2. You can test the code at the localhost. However, to test the timeout, you can intentionally drop some chunks from the receiver, and do not send back an acknowledgement.

Sample code for SIGALRM

```
=====================================================================
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define SLEEP_VAL 5

static int alarm_fired = 0;

void mysig(int sig)
{
    pid_t pid;

    printf("PARENT : Received signal %d \n", sig);

    if (sig == SIGALRM)
    {
        alarm_fired = 1;
    }
}

/*  In main, we tell the child process to wait for five seconds
    before sending a SIGALRM signal to its parent.  */

int main()
{
    int pid;

    printf("PARENT : Alarm application starting\n");
    printf("PARENT : Sleep time %d\n", SLEEP_VAL);

/* Instead of using fork, we are using the alarm function here
   No child process needed */

    alarm(SLEEP_VAL);

/* The parent process arranges to catch SIGALRM
   with a call to signal and then waits for the inevitable.  */

    printf("PARENT : Now waiting for alarm to go off\n");

/* Upon receiving SIGALRM, call function mysig()
   Note : SIGALRM not yet called  */

    (void) signal(SIGALRM, mysig);
do
    {
        sleep(1);

        printf("PARENT : Still waiting...\n");

        // Wait for any signal call
    }while(!alarm_fired);

    printf("PARENT : Alarm! Ding Ding!\n");

    exit(0);
}
```