CPE464 - Programming Assignment #3 – Selective Reject
Spring 2025 Due: See canvas for due dates

Extra Credit: See canvas for extra credit date. If you turn in your program before the extra credit deadline you will receive 25% extra credit (Your program must be fulling working including the multiprocess part of the assignment.)

Overview:
In this assignment you are to write a remote file copy using **UDP**. To do this you will implement two programs in C (or C++). These are **rcopy** (the client program) and **server**. The **rcopy** program takes two filenames, a from-filename and a to-filename, as parameters. The transfer of the files is for server to download the from-file to rcopy. rcopy creates the output file (the to-filename), receives the data packets from server and writes this data to the to-filename.

To be clear, in this assignment, server is sending the file to rcopy. This is a download from the server to rcopy. In this document:

- sender or sending process is referring to the server
- receiver or receiving process is referring to rcopy

This program will use **a sliding window flow control algorithm using selective-reject ARQ.** This program must use **UDP**. You will need to break the file up into packets and append an application-level header you create to each packet as described below (e.g you are creating an application PDU).

In order to induce errors you will **not** use the normal sendto(…) function. Instead, you will use a sendtoErr(…) function. This function will periodically drop packets and flip bits in your data You may **not** use the normal sendto(…) **anywhere** in either (rcopy/server) program. To catch bits flipped by the sendtoErr() function you will use the Internet Checksum to calculate a checksum to put into your application-level header as detailed below.

Some Coding requirements:

1) **Poll library**: You must use poll() to facilitate flow control. You can write your own library (and it must be in separate files) or you can use the poll library provided by Prof. Smith.
2) **Window/Buffering Library**: You **must** implement your windowing on the sender and buffering on the receiver as a library (or multiple libraries) with an API in a separate .h/.c file(s). On the sender this is windowing (e.g. keeping track of current, upper and lower), on the receiver of the data it is just buffer management to store data when packets are lost.

   This means that the window and buffer data structure, and accessor functions must be defined in a separate c (c++) file(s) and .h file(s). These must be separate from your other code. Both your client (rcopy) and server must use this code (your library code) for managing the window/buffered data. You are allowed to have up to (at most) 5 global variables in this file(s), but these global variables cannot be accessed by code in any other file. The only code that may be in these C/C++ file is your window/buffering library code. All access to your windowing data/buffering data must be via your library's accessor functions.

   You must implement your windowing/buffering functionality using a circular buffer you implement, and you cannot use a preexisting data structure (e.g. cannot use something provided by C++). Your window data structure/buffer data structure must be a malloc()ed **array** of structures and managed as a **circular buffer/queue** (e.g. index into window = packet-sequence-number % window size).

   Note – if you directly access any of your window/buffer data structure in code other than the library file, your program will not be accepted.

Programs to write (rcopy and server):

1) **rcopy**: This is the client program. This program is responsible for taking the two filenames as command line arguments and communicating with the server program to download the from-file to the to-file created by rcopy. The rcopy program will be run as:

   rcopy ***from-filename to-filename window-size buffer-size error-rate remote-machine remote-port***
   (e.g. rcopy file1 file2 10 1000 .2 unix2 44444)

   where:

from-filename:	is the name of the file to download from the server to rcopy
to-filename:	is the name of file created by rcopy (rcopy writes to this file)
window-size	is the size of the window in PACKETS (i.e. number of packets in window)
buffer-size:	is the number of data bytes (from the file) transmitted in a data packet[1]
error-rate	is the percent of packets that are in error (floating point number e.g .1)
remote-machine:	is the remote machine running the server
remote-port:	is the port number of the server application

2) **server**: This program is responsible for sending the requested file to rcopy. This program should never terminate (unless you kill it with a ctrl-c). It should continually process requests to download files to rcopy. To receive full credit the server must process multiple clients simultaneously.

   The server needs to handle error conditions such as an error opening a file by sending back an error packet (e.g. error flag) to the rcopy program.

   The server will receive the **window-size, buffer-size and from-filename** from rcopy for each file exchange.

   The server should output its port number to be used by the rcopy program. The server program is run as:

   server ***error-rate*** **[*optional-port-number]*** **(e.g. server .1 44444)**

   where:

error-rate	is the percent of packets that are in error (floating point number)
optional-port-number	allows the user to specify the port number to be used by the server (If this parameter is not present you should pass a 0 (zero) as the port number to bind().)

Requirements:

1) Do not use any code off the web or from other students. Do not look at any other code that provides a solution to this problem or parts of this problem. You cannot use ChatGPT or any other AI assistant to write or help write your code. The work you turn in must be your entirely your own. You may make use of the code I handed out in lecture, any code I provided and the Internet Checksum code.

2) Both rcopy and server should call the sendtoErr(…) function for all communications between rcopy and server (e.g. for connection setup, data, RR's). You should use the sendtoErr function for all packet

[1] This buffer-size is the number of bytes you should **read** from disk and send in each packet. This is not the packet size. The packet will be bigger (since it needs to include at least an application PDU header.) The last packet of file data may be smaller and file name exchange packets (or similar control packets) must be no bigger than needed to communicate the required information.

transmissions including transmitting the file name to the server. You should **not** use the normal sendto(…) function in your program.

3) You must provide a README file that includes your first and last name and your lab section time (9am, noon, 3pm, no lab).

4) You must use the poll() call (e.g. my poll library) on both the client and server to facilitate flow control. You may directly use my pollLib (recommended). If you want to write your own poll library that is fine but it must be in a separate file.

5) In this program you should never call recvfrom() (except in the server when receiving on the main server socket) without first calling poll(). If you find yourself thinking it would be ok to call recvfrom() without first calling poll(), your design is probably incorrect or your understanding of the program is probably incorrect. Ask for help! Note – the only exception is the recvfrom() call in the main server process waiting to recvfrom() the first packet from a client.

6) If the sender is not able to open the from-file, rcopy should print out the error message: **Error: file <*from-filename*> not found**. rcopy should then exit.

7) If the receiver cannot open the to-file, rcopy should print out the error message: **Error on open of output file: <to-filename>**.

8) You must use an application PDU header in all of your packets. The header must consist of a 32-bit sequence number in network order, the internet checksum and a one byte flag field. So, your packet format for ALL packets should be packet seq#, checksum, flag, and then data/Filename/whatever.

9) The data payload (the data from the file is the payload) will be from 1 to 1400 bytes. This means the smallest application PDU containing file data is 8 bytes (7 bytes of application header and 1 byte of data) and the maximum PDU size will be 1407 (7 bytes for the header and 1400 bytes of file data).

10) The window size will be less than 2^{30}.

11) Regarding the format for the **RR and SREJ packets** (it is a normal header created by the sender of the RR/SREJ and then the seq number being RRed/SREJed.) This means the application PDU size for a RR/SREJ packet is 11 bytes (7 bytes of header + 4 bytes for the sequence number you are RR'ing/SREJ'ing). The seq number being RR/SREJ must be in network order.
    - **RR PDU format:** Packet Seq number, checksum, flag, RR seq number (what you are RRing)[2]
    - **SREJ PDU format:** Packet Seq number, checksum, flag, SREJ seq number (what you are SREJing)

12) You must use the following flag values in your header. If there are other types of packets you want to send (meaning you need other flag values) that is ok. Please document any additional flag values in your readme file. You must use the following value:

    a. Flag < 32 if not specified in this list cannot be used, if you create your own flag use ≥ 32
    b. Flag = 5 RR packet
    c. Flag = 6 SREJ packet
    d. Flag = 8 Packet contains the file name/buffer-size/window-size (rcopy to server)

[2] The packet sequence number is the sequence number created by the sender of the RR/SREJ. Both rcopy and server maintain their own set of sequence numbers and this is the number put into the first 4 bytes of the header.

e.	Flag = 9	Packet contains the response to the filename packet (server to rcopy)
f.	Flag = 10	Packet is your EOF indication or is the last data packet (sender to receiver)
g.	Flag = 16	Regular data packet
h.	Flag = 17	Resent data packet (after sender receiving a SREJ, not a timeout)
i.	Flag = 18	Resent data packet after a timeout (so lowest in window resent data packet.)
j.	Flag ≥ 32	If you wish/need to create other flags use ≥ 32

13) Your programs needs to be written such that rcopy **ends first** after the file transfer has been completed. So if the final response packet from rcopy back to server is lost, rcopy needs to end and it is up to server to keep waiting/trying. In your design, your tear down after the transfer has completed flow needs to have rcopy send the last packet to the server.

14) If no packets are lost during the final packet exchange, the server child process should end immediately after receiving the final response (note it is assumed that rcopy will terminate after it sends its final response to the server). So the server cannot be written so that it <u>always</u> sends the last packet 10 times or it <u>always</u> hangs around for 10 seconds. So if no data is lost, your shutdown of the server child process has to be clean.

15) Remove any debugging print statements before you handin your assignment.

16) If you resend a packet **10 times** you can assume the other side is down and terminate/end the process gracefully.

17) Your solution for sending the filename should not permanently hang a server child process nor should it cause rcopy to terminate just because of a few lost packets. Your solution needs to handle the loss of the filename and the loss of the filename acknowledgement up to **10 times**. This means rcopy should transmit the filename up to 10 times before quitting. You cannot blast out the filename multiple times, but can only resend the filename if a timeout occurs.

18) When sending file data and **your window closes** on the sender you should poll for 1-second (1000 milliseconds) waiting on RRs. On the sender, only if its window closes can you call poll() with a non-zero (e.g. 1000 milliseconds) value.

19) If you are sending data and your window is open you should process RRs/SREJs but you should NOT block on poll (so call poll() with 0 milliseconds) So while your window is open, send one packet, call poll(with 0 milliseconds) to see if any RRs/SREJs are available, if RRs/SREJs are available then recvfrom() and process all of them.

20) When sending file data, if your window closes on sender (meaning you have sent an entire windows worth of data) and then your 1-second poll times out on the sender, you break this deadlock by resending the lowest PDU buffered in your window (and only the lowest packet). If your receiver receives a packet lower than expected it should respond with the highest RR/SREJ allowed.

21) Regarding buffering packets for your window processing. You cannot buffer the entire file on either the server or rcopy. You can only maintain buffers of the size of your window. Your buffer must be a normal C array (it can be an array of structs but must be an array). You cannot use any data structures built into the language or supplied in 3rd party libraries. Your window management must be done by you. Also, you cannot move back and forward in the disk file instead of buffering. Any data held for window processing must be buffered in your array. Any code to process this buffer must be written by you.

22) The maximum filename length is 100 characters. rcopy should check the filename lengths prior to starting the file transfer. Print out an appropriate error message if a filename is too long and terminate rcopy.

23) You cannot call sleep or usleep anywhere in your program.

24) The program names, rcopy and server must be used. Also the run-time parameters should be in the order given. Name your makefile: Makefile.

25) Use datagram sockets for these programs (UDP). Your program must be written in C (C++). Also, while some protocol stacks do support UDP calls to connect() and accept() – you may not use these functions in your programs.

26) No process (i.e. rcopy, server child) should permanently block when the other side terminates. For lost data, the sender of the data (eg. server sending file data) should try to resend packets 10 times using a 1-second timeout. The receiver of the data (e.g. rcopy during the file data transfer) should use a 10 second timeout while waiting for data.

27) The main server process should not terminate. It should loop waiting to process new rcopy requests. The server should only terminate if it receives a ^c.

28) When sending file data, the only time the sender should block (on poll for 1 second) is when the window is closed (meaning you have sent all the data you can.) Otherwise, the sender should not block. Use a non-blocking poll(…) (which means set your time value in your poll call to 0 (zero)) to check for RR's/SREJ's when the window is open).

29) You **do not need to use** the sliding window concepts for the filename exchange or after you send the last packet. Sliding windows only makes sense during the file data exchange.

30) When you are sending data and your window is open the flow should be: While window is open, read a packets worht of data from disk, send one data packet, do a non-blocking check for RRs/SREJ's and if RR's/SREJ's are available process all of them, goto while window is open.

31) Your server must handle multiple clients at the same time via **multiprocessing (fork).** (You will lose 25% if this does not work.)

32) When using fork(), you must call sendtoErr_init() (with the random seed on) in each child before the child sends a packet. This means on the server you need to call sendtoErr_init() in both the parent process and in each child process.

33) Summary
    a. You can **only** resend lost/corrupted data. You cannot resend data that was received correctly unless you need to recover from a timeout (and then only 1 packet).
    b. You must send a SREJ on lost data (you cannot just wait until the other side times out. But you only need to send a SRJE once per lost packet – let your timeout recover from a lost SREJ.)
    c. If your window is closed and you timeout (so you have not heard about an entire window of data and have waited 1 second), send the lowest unacknowledged (so un-RRed) packet and wait for a response. The intent of this packet is to wake up (break the deadlock) the other side and hopefully open up your window. Continue this for 10 tries (wait 1 second, send lowest packet) or until you get an RR or SREJ.
    d. Do **not** set timers for every packet. That would be a major pain.
    e. Your blocking 1-second poll() may only time out if:
        i. All RRs for a window are lost

ii. or all data in a window is lost
iii. or A SREJ packet is lost
iv. or the recovered packet from a SREJ is lost
v. You cannot timeout during the data transfer in any other situation

f. The sender should process RRs/SREJs (without blocking) after every send(). For the poll() call, if you set the time value to 0 (zero) poll() will check the socket for data but return immediately (so non-blocking).

a. On sender during the data exchange, you can only call poll() with a time value greater than 0 if the window is closed. If the window is open you can only call poll() with a zero time value.

b. You cannot resend good data, if a packet arrives in the current window and is not corrupted it is regarded as good data and should either be buffered or written to disk.

c. You may **not** go back/forth on disk. You cannot use any seek (e.g. lseek(), fseek()) function call. All data that needs to be buffered for windowing must be done using a circular buffer in memory.

A run of these programs would look like:

On unix1 (sever using a 10% error rate):

```
> server  .1
Server is using port 1234
```

On unix3 (using a window size of 10, buffer size of 1000 bytes, client side 10% error rate):

```
> rcopy myfile1 myfile2 10 1000 .1 unix1.csc.calpoly.edu 1234
  Error: file myfile1 not found.
>rcopy myfile4 myfile5 10 1000 .1 unix1.csc.calpoly.edu 1234
>
```

On unix2 (at the same time as unix3) (using a window size of 5, buffer size of 900 bytes, client side 5% error rate):

```
> rcopy myfile7 myfile8 5 900 .05 unix1.csc.calpoly.edu 1234
>
```