CPE464 - Programming Assignment #3 – Selective Reject
Due Friday May 28, 2021

This program is due on **Friday, May 28th at 11:59 pm.** If your program is late you will lose 15% per day. The last day you can turn in this assignment for any credit is Sunday May 30th at 11:59 pm.

Extra Credit: If you turn in a complete working program before 11:59 on Sunday, May 23rd you will receive 25% extra credit (This includes getting the multithreaded/multiprocess part of the program working to receive the extra credit.)

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 rcopy to download the from-file from the server. rcopy creates the output file (the to-filename), receives the data packets from server and writes this data to the file.

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 your own 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.[1] 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.

Some Coding requirements:

1) **Poll/Select library**: You **must** use a library (similar to program #2) for your calls to select()/poll(). This library should allow you to add and remove sockets from you file descriptor set and call the select()/poll() function. Just use whatever library you created/used in program #2 (e.g. my pollLib).
2) **Window Library**: You **must** implement your windowing functionality as a library. On the sender this is windowing (e.g. keeping track of current, upper and lower), on the receiver its more just buffer management to store data when packets are lost.

   This means that the window/buffer data structure, and accessor functions must be defined in a single c (c++) file (and single .h file) separate from your other code. Both your client (rcopy) and server must use this code (your library) for buffering of data in a window. You are allowed to have up to (at most) 3 global variables in this file, but these global variables cannot be accessed by code in any other file. The only code that may be in this C/C++ file is your window library code. All access to your windowing data must be via your window library's accessor functions.

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

[1] Just to be clear, the *from-file* is being downloaded from server and being put into the *to-file* on the client (rcopy). This is a download of the file from the server to rcopy.

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 from the server to the to-file created by rcopy. The rcopy program will be run as:

   rcopy ***from-filename to-filename window-size buffer-size error-percent remote-machine remote-port***

   where:

from-filename:	is the name of the file to download from the server (server sends this down)
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[2]
error-percent	is the percent of packets that are in error (floating point number)
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 downloading 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 the file to download by sending back an error packet (e.g. error flag) to the rcopy program.

   The server will receive a **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-percent optional-port-number***

   where:

error-percent	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. 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] 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 a 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.

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 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).

4) You may use either select() or poll() for this program. Your select/poll functionality must be in a separate file (e.g. it's a library like my pollLib). You may directly use my pollLib.

5) If the server is not able to open the from-file it must send back an error to rcopy. rcopy should print out the error message: **Error: file <*from-filename*> not found on server**.

6) If rcopy cannot open the to-file, rcopy should print out the error message: **Error on open for output of file: <to-filename>**. Then print out the perror() and exit. This could happen the disk is full or the rcopy does not have write access to the directory/file.

7) Your programs need 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 try 10 times before terminating.

8) If no packets are lost during the final packet exchange, server should also 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 always sends the last packet 10 times or it always hangs around for 10 seconds. You need to have a clean shutdown process.

9) In this program you should never call recvfrom() (except in the server when receiving on the main server socket) without first calling select()/poll(). If you find yourself thinking it would be ok to call recvfrom() without first calling select()/poll(), your design is incorrect or your understanding of the program is incorrect. Ask for help!

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

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

12) Your solution for sending the filename should not permanently hang a server process/thread 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.

13) When sending file data and your window closes on server you should select/poll for 1-second waiting on RRs. If you are sending data and your window is open you should process RRs but you should NOT block on select/poll (use a non-blocking select/poll... so…while your window is open, send one packet, see if any RRs are available, if so process them, repeat the loop).

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

15) 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.

16) The buffer size will be from 1 to 1400 bytes. The window size will be less than 2^{30}.

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

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

19) You must use a header in all of your packets. The header should have as a minimum 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, data/Filename/whatever.

20) Regarding the format for the RR and SREJ packets (it is basically a normal header created by the sender of the RR/SREJ and then the seq number being RRed/SREJed.) This packet seq number must be in network order.
    - **RR packet format:** Packet Seq number, checksum, flag, RR seq number (what you are RRing)[3]
    - **SREJ packet format:** Packet Seq number, checksum, flag , SREJ seq number (what you are SREJing)

21) 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 = 1	Setup packet (rcopy to server) – optional
b.	Flag = 2	Setup response packet (server to rcopy) - optional
c.	Flag = 3	Data packet
d.	Flag = 4	(Not used this quarter)
e.	Flag = 5	RR packet
f.	Flag = 6	SREJ packet
g.	Flag = 7	Packet contains the file name/buffer-size/window-size (rcopy to server)
h.	Flag = 8	Packet contains the response to the filename packet (server to rcopy)
i.	Flags > 8	should be used for other things (e.g. EOF, ending the connection)

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

[3] 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.

23) 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.

24) No process (e.g rcopy, server child/thread) 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 after 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. This allows rcopy to terminate cleanly if it no longer can communicate with the server.

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

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

27) You will 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.

28) When you are sending data and your window is open the flow should be: Send one data packet, non-blocking check for RRs/SREJ's and if RR's/SREJ's are available process all of them, goto send one data packet.

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

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

31) 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 select(..)/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. You cannot timeout during the data transfer in any other situation

f. The server should process RRs/SREJs (without blocking) after every send(). In select() or poll() if you set the time value to 0 (zero) select()/poll() will check the socket for data but return immediately.

a. You can only call select()/poll() with a time value greater than 0 if the window is closed. If the window is open you can only call call select()/poll() with a zero time value.

b. On server, when the window is closed you should do a blocking call select()/poll() for a time of 1-second.

c. 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.

32) A run of these programs would look like:

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

```
> server  .01
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 .01 unix1.csc.calpoly.edu 1234
  Error: file myfile1 not found on server.
>rcopy myfile4 myfile5 10 1000 .01 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
>
```