PORTLAND STATE UNIVERSITY
CS494/594: INTERNETWORKING PROTOCOLS

PROGRAMMING PROJECT PART 2
DUE: March 7th, 5:00 PM

1. Development Setup

For this programming assignment you will work in the CS Linux Lab (linuxlab.cs.pdx.edu) located in FAB 88-09 and 88-10. If you don't already have an account, go to http://www.cat.pdx.edu/students.html for instructions.

For this Project you must **use C/C++, UDP Sockets, POSIX Threads, Mutex/Semaphores, POISIX Timers and Make**. You are allowed to use any interfaces defined by C++11 or C++14. Please refrain from any other languages or 3rd party libraries. We will use the GNU C/C++ Compiler (gcc/g++) and Make already installed in the development machines in the lab.

Grading will be done in this setup so please make sure that your code works under this conditions.

2. Problem Description

For the second part of the class project we will design and implement an upgraded version of the simple File Transfer Protocol from Part 1**. Our focus will be on improving the performance of the reliable channel and implementing flow control and congestion control mechanisms.**

You will implement both the Client and the Server applications for our Simplified Remote Copy Protocol. As with Part 1, your client must use command line parameters to allow the user to specify the IP address, port, and file to transfer. .This is an example of the command a user might use to download a file located in the server computer at 120.45.67.126 and listening on port 1080:

./cs494rcp_client 120.45.67.126 1080 /Docs/Files/myfile.txt

The server must use the command line parameters to allow the user specify the port to listen for connections from other clients and it must also **allow the user to specify the maximum window size as a command line parameter**. This is an example of the command a user might use to start the server listening on port 1080 and using a sending window size of 20 packages:

./cs494rcp_server 1080 20

Also, as part of the project, your server program should print statistics about the performance of Go-Back-N after the file has been successfully sent: **the number of data packets retransmitted, total data packets sent and the time elapsed to send the entire file.**

3. Go-Back-N

For the second part of the project we will implement the **Go-Back-N flow control protocol**. We will modify our stop and go implementation from Part 1 to allow to send multiple packets and pipeline the acknowledgments to improve throughput. Go-Back-N is a sliding-window based protocol, that is, the protocol operates over a limited but configurable number of packets.

In Go-Back-N the server sends up to N packets, for a window of size N, before waiting for an ACK from the client. Furthermore, in Go-Back-N acknowledgments are **cumulative**, hence a single ACK can mark several packets as successfully received. More specifically an Acknowledgement for packet N+1, will also acknowledge packet N. Therefore this opens the possibility that the client might decide to send only acknowledgments for some of the packets. **However, in our implementation, the client will still acknowledge every single packet.**

Once the ACK from a data packet at the beginning of the window is received, the server can slide the window allowing for an additional packet to be sent. Please note that **Go-Back-N is a pipelined protocol and therefore the server must send data and receive acknowledgements in a concurrent fashion. Therefore, is not sufficient to simply send N packets and then wait for N acknowledgments in the same thread.** Instead your server implementation will need separate threads for sending data and receiving ACKs.

Go-Back-N still requires packets to be received in order. The client will drop any packets that are not the expected packet. This makes the client behave very similar to Stop-And-Go and hence much of the work is performed by the server.

To recover from packet loses, Go-Back-N uses a timer mechanism in which the oldest packet in the window must be acknowledged before the timer expiries. In case that the oldest packet in the window expires the server will re-send every single packet in the window. Please note that this might cause the client to receive several duplicated packets. This is normal and an accepted behavior in Go-Back-N.

Go-Back-N can be used with a static window size or a dynamic window size. We will use a configurable static window size specified by the user when launching the server.

4. Implementation Overview

You are allowed to use only one socket connection between client and server to send and receive packets. This is important as our protocol is a pipelined protocol and not a parallel version of Stop and Go.

Many of the components of Part 1 will remain unchanged: Your program still needs to perform a three-way handshake to establish the connection before sending the file and it must perform a simple two-way handshake to close the connection after the file has been sent. Also most of the client will remain unchanged. However, data transmission over the server will change significantly.

These are a few key concepts you must address as part of your implementation:

1) Your **server must send data and receive acknowledgements in a concurrent fashion**. Therefore you need to add a separate thread to receive acknowledgements. The main thread in the server will still be responsible for sending packets.
2) You must implement a sliding window/ queue that allows the server to keep the state of each of the data packets (i.e. Available for Sending, Sent, Acknowledged, Expired). Please note that the queue must not hold the entire file. Your queue must old at most N packets, where N is the size of the sliding window specified by the user. You are allowed to use one of several STL C++ containers including but not limited to maps and vectors, however, you can also implement your own circular queue.

3) Similarly, to Stop-Wait ARQ from Project 1, your server must **wait for a timer to expire in case the packet is lost**. If no ACK has been received before the timer expires it means the packet was lost in transit and must be resent.
4) Our pipelined design, leaves our sending thread without work when the queue is full. Your design must address this case by putting the thread to sleep until there is at least one slot available in the sliding window. For this purpose you can use a Conditional Variable, a mutex or a semaphore. **However, you must not spin while the sliding window / queue is full**
5) Finally, you must address the data races created by the 3 concurrent execution flows (i.e. Main/Sending Thread, Acknowledge Thread and Timer). You must use a lock (i.e. mutex) to protect the critical section connection and exit.

5. Unreliable Channel Simulator

To aid in your testing we have provided you with a packet simulator that will drop, delay and duplicate packets. The simulator works by interposing the POSIX sendto() and rcvfrom() functions during compile time. Therefore, to use the simulator, you must compile the provided C++ source file **unrel_sendto.cpp** as part of your project and link against both your client and server. Moreover, the linking process requires to pass the following flags to the ***-Wl,-wrap=sendto -Wl,-wrap=recvfrom***

For example to compile the program server.cpp and link the simulator you can use the following command:

```
g++ -Wl,-wrap=sendto -Wl,-wrap=recvfrom server.cpp unrel_sendto.cpp -o server
```

As an indication that the simulator was properly enabled, your client and server should print a message with the parameters of the simulator as soon as they start.

6. Testing Script

To aid in grading and testing your project, you must prepare a BASH script (test.sh) that starts the server on port 1080, then starts the client on port 1080 and send the file “test.jpg” provided as part of the project package. The script then must compare the input and output files for correctness. A good way to do this is by using md5sum.

The project package provides a sample BASH script you can use to write your own script.

7. Hand-In

For submission, you should provide only source code (*.c, *.cpp, *.h) and a Makefile script that compiles both the Client and the Server code and a test script test.sh.

Please pack your files into a TAR file before submitting your solution. Do not include test.jpg, this PDF, executables or object files. We will use the D2L system for submission (https://d2l.pdx.edu/). Please remember that there is no late policy.