CS 5450: Networked and Distributed Systems
Homework 1: Go-back-n (Spring 2020)
Instructor: Vitaly Shmatikov, TAs: Jialing Pei, Eugene Bagdasaryan

Due: 11:59PM February 19, 2020

About this assignment

This should be a fun project with images and TCP/IP stack.

You can work in groups of 2, there is a Piazza post that will help you find teammates. There are two slip days **for the entire course** that you can take on this or any other assignment. If you run out of slip days and submit late, a penalty of at least 50% of the grade will be assessed for that assignment. Be sure not to submit late – we will enforce a strict 11:59 PM deadline.

Use Piazza for questions. The TA will do his best to respond twice a day to questions but peers may well respond faster and you might see answers to your questions even before you know what those questions are :}

We encourage you to start early and leave some time for writing up your report.

Useful links

Website: `https://cs5450.github.io`
Piazza: `https://piazza.com/cornell/spring2020/cs5450/home`

Getting started

- Download the skeleton code from CMS.
- From the Linux command line type:

```
gunzip HW1_skeleton.tar.gz
tar -xvf HW1_skeleton.tar
```

- The above commands will extract the skeleton files for you.
- To compile the code, simply type `make` at the command line.

Introduction

In this project you will be implementing a Go-back-n (GBN) protocol. Real-world systems are usually connected by unreliable links, which can reorder, lose, or disrupt the packets exchanged. The task of your transport layer implementation is to make sure that data sent on one end appears on the other end exactly as it was sent. You will not be implementing the reliable transport protocol in a simulated environment, but rather in real-world UNIX systems.

Your implementation should address the three above-mentioned issues and should provide to the application the illusion of having a byte stream between the two communicating processes. You are not asked to provide full-duplex data transmission (i.e., both ends transmit data packets to each other), but you must implement a simple congestion control mechanism. When packets are dropped, the protocol will start sending at a slower rate (e.g., Go-back-1), and speed up when it successfully receives an acknowledgement (e.g., Go-back-2, Go-back-4). In our protocols, only one end sends data; the other end only sends acknowledgments back to the sender.

There is skeleton code available on CMS to help get you started. The sender and receiver files will not need to be modified for your project. These exact commands should run the program:

```
./sender <hostname> <port> <filename>
./receiver <port> <filename>
```

Requirements

You should implement the following functions for this project:

- `gbn_socket()`: used to setup a socket.
- `gbn_connect()`: used to initiate a connection.
- `gbn_send()`: used to send packet data using GBN protocol.
- `gbn_recv()`: used to receive packet data using GBN protocol.
- `gbn_close()`: used to end a connection and close the socket.
- `gbn_bind()`: used to bind a socket to your application.
- `gbn_listen()`: used to change state to listening for activity on a socket.
- `gbn_accept()`: used to accept an incoming connection.

In some cases your function may not need to provide any additional functionality beyond that of the system call; in this case, all you will need to do is make the system call (passing any necessary arguments). You must build your protocol using the system calls provided by UDP to applications:

- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
- `bind()`

It is highly recommended that you read the manual pages for these system calls. You can use Beej's guide (`http://beej.us/guide/bgnet/`) or online sources.

Note: You must use only the five UDP system call functions mentioned above to perform network operations. For example, it is not ok to use such system calls as `send()` in place of `sendto()`. Implement additional helper functions as necessary.

Specifications

You can find the description of the Go-back-n protocol **here**. It's part of Chapter 3.4.3 of the textbook by Kurose and Ross "Computer Networking: A top-down approach" (it's a recommended textbook if you want to learn more about computer networking). Here is a cool visualisation:
`http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/`

The protocol you are about to implement in this project is a simplified version of the protocol described in the textbook.

For implementing the time-outs mentioned in the protocol description, you need to set up timer(s) for the packets you send (a single timer or multiple timers depending on which protocol you use). You can do that using either the `alarm()` or the `setitimer()` functions. These functions take one argument which represents the amount of time it should wait before sending a signal to the process. If the timer expires, a `SIGALRM` signal is sent to the process. When the process receives such a signal, it stops running the current code and runs the signal handler. The signal handler must be explicitly assigned to a user-defined function in the code. It can be done using the `signal()` function. This function takes two arguments: the signal type (`SIGALRM` in our case) and a function pointer (the name of the function *you* define to handle the signal), and establishes the binding between the two.

Besides running the signal handler, signals can be useful to wake up a sleeping process. For instance, whenever we call the `recvfrom()` function to read a packet from a UDP socket, the process goes to sleep until a UDP datagram arrives for that socket. If the timer expires while we are still waiting for a packet to arrive, the process is awoken. The first code it runs after waking up is the signal handler. After the signal handler returns, the function that caused the sleep (in our example, `recvfrom()`) returns and the regular execution of the process resumes. The return value of that function can be checked to see if it returned due to a timer expiring or due to actually receiving a UDP datagram. For example, whenever it is awaken by the timer, `recvfrom()` returns -1 and a special error variable called `errno` is set to `EINTR`, which means "interrupted function call". When `recvfrom()` is awoken by a packet being received, it simply returns the number of bytes received. This difference can be used to make the retransmission part easier to implement.

You should design a simple state machine for your protocols and implement it. Your protocols should have a connection setup and a connection teardown. In the connection setup, a so-called `SYN` packet must be sent to the server to initiate the connection. The server then replies with a `SYNACK` packet to the client if it accepts the connection or with a `RST` packet if it rejects the connection. In the connection teardown, the party wanting to finish the connection should send a `FIN` packet to the other end host. This other host, in turn, replies with a `FINACK` packet to acknowledge the first `FIN` packet.

Your Go-back-n algorithm will have a basic congestion control mechanism. To be more specific, you need to dynamically adjust your transfer speed(i.e., the value of n) as the network traffic situation traffic changes. As a hint, we recommend you to try a base speed $1(2^0)$ at first and try to adjust it exponentially according to network traffic situation. Note that the value of n should be the power of 2 so the speed will take the form of 2^t where t is the speed ratio you need to adjust. All you should do is to increase t when traffic is light(all acks are well received within the timeout) and decrease t when traffic is heavy(at least one ack lost after the timeout). One thing to be mentioned: You should not set your speed way too large so that the size of packets you need to keep track of exceeds the size of your buffer pool. In order to evaluate your algorithm's performance, We will offer **extra bonus points for the fastest file transfer**.

In the skeleton code provided, we provide a header for the packets composed of a 16-bit type field and a 16-bit checksum field. The type field identifies the type of packet being sent (e.g., `SYN`, `SYNACK`, `FIN`, etc.). The checksum field is used to check the integrity of the received packet. It is calculated by the packet sender and checked by the packet receiver. The checksum should take into account the entire header and the carried data. We provide the function that can be used to both calculate and check the checksum.

The `SYN`, `SYNACK`, `FIN`, `FINACK`, and `RST` packets contain no `DATA` field. They are only composed of the type and checksum fields. In your code, we also suggest adding a few states to your state machine, such as `CLOSED`, `SYN SENT`, `SYN RCVD`, `ESTABLISHED`, `FIN SENT`, and `FIN RCVD`, but you are welcome to change them if you want.

In order to test if your protocol works with losses/corruption, you should not call the `recvfrom()` or `sendto()` function directly. Although unreliable, data transmission in local networks can have little loss and almost no corruption. Instead, you should use wrapper functions we provide called `maybe_recvfrom()` and `maybe_sendto()`. Both functions select packets at random to be dropped/corrupted based on some fixed loss and corruption probabilities. They have the same parameters as the regular `recvfrom()` and `sendto()` function used to receive UDP datagrams.

For this project, you can safely use static values for the timer(s); that is, the time interval to wait after a packet is sent is not dynamically adjusted. It is a predetermined value that does not change throughout the connection (a value of 1-2 seconds should work well for the project). You can also assume that if the same packet is sent five times without receiving any acknowledgment, the connection is broken. You can compare the original file to the received file using simple UNIX commands, such as `diff file1 file2` or `md5sum file1 file2`.

Deliverables

Note that the programming language that you use should be C and **your program should run on a Linux machine with Ubuntu OS**. **As our autograder runs on Ubuntu 16.04, we strongly recommend to use the same setup**. Some of the previous submissions that were developed for MacOS didn't work and were deducted points.

- You need to submit all **commented** source files in a single tar.gz file. Type `make tarball` at the command line and submit the single tar.gz file to CMS.
- You are also required to submit a short report discussing the results (1-2 pages). The printed reports should be turned in CMS by the same deadline. Particularly, you need to include the following items in your report:
  - A general description of the protocol
  - A paragraph about the performance optimization under different network reliability assumptions

The project due date is 11:59PM February 19, 2020. Good luck!