
CS 5450: Computer Network Fundamentals
Homework 1: Go-back-n (Fall 2018)
Instructor: Professor Vitaly Shmatikov, TA: Eugene Bagdasaryan

Due: 11:59PM Wed, Oct 3 2018

Note

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

You can work in a group of 2, there is a Piazza post that will help you find teammates. There are two slip days that you can take, however if you miss your deadline a penalty of at least 50% of the grade will be applied. Please make sure not to be late for the submission – 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.

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 (Go-back-1), and speed up when it successfully receives an acknowledgement (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 in page 230 (chapter 3.4.3) of the textbook by Kurose and Ross "Computer Networking: A top-down approach", it is possible to find a PDF version online or take it in the library. There is another 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. **You will only need to consider window size of 1, 2 and 4 for your implementation.**

For implementing the time-outs mentioned in the protocol descriptions, 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 awoken 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. It will switch between three modes: fast mode ($n = 4$), moderate ($n=2$) and slow mode ($n = 1$). In slow mode, the next packet will not be sent until the sender receives acknowledgement for the previous packet. In moderate mode, a second packet may be sent before receiving an acknowledgment for the previous

packet. The fast mode allows to send four packets. The protocol will start in slow mode. Whenever it is in slow mode and receives an acknowledgment for the most recently sent packet, it will switch to moderate mode and after success on moderate mode will switch to fast mode. When it is in fast or moderate mode and a packet needs to be retransmitted because the sender did not receive acknowledgement from the receiver, the program switches directly to slow mode. We advise you to first implement slow mode, then moderate and slow mode, and after your solution works, then implement switching between three modes.

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()` function directly. Although unreliable, data transmission in local networks can have little loss and almost no corruption. Instead, you should use the wrapper function we provide called `maybe_recvfrom()`. The `maybe_recvfrom()` function selects packets at random to be dropped/corrupted based on some fixed loss and corruption probabilities. It has the same parameters as the regular `recvfrom()` 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. As our autograder runs on Ubuntu, 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 tricky parts of the implementation

The project due date is 11:59PM Oct 3, 2018. Good luck!