# GBN-socket

> Homework 1 - CS5450 Networked and Distributed Systems
>
> Chenyu Zhang - Cornell ID 4982538 - cz442@cornell.edu
>
> Shanwen Wang - Cornell ID xxxxxxx - swxxx@cornell.edu

## Intro

Implement a Go-back-n (GBN) protocol, which is used to describe a TCP-like data transmission mechanism.

It should support following features.

1. **Reliable transmission** on unreliable links, which can *reorder*, *lose* or *disrupt* the packets exchanged.
2. **Simple congestion control**, like *adjustable sliding sending window*.

p.s. Don't need to implement full-duplex communication. Channel on a single direction is fine.

## Tasks

Finish the following functions:

- `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.

We can only implement this protocol based on UDP socket in C. In other words, we can't use other socket calls provided by OS except:

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

## Code & Test & Some Modifications

1. We tried our best to write clear code that's easy to understand, with comments under every

important code block.
2. We changed the compiling option from `-std=ansi` to `-std=gnu99`.
3. We added several `printf()` functions to the code, which helped us debug and test. For I/O is not so frequent, it will not affect the speed of transmission that much.
4. We compiled it on *Ubuntu 18.04 x86_64 GNU/Linux* environment provided by AWS, and successfully transmitted 5 files which were all around 90MB. MD5 of source and target were the same.
5. Changed a little bit of sender.c, like what's mentioned in Piazza. (h_addr -> h_addr_list[0])

# Protocol Description

## A Typical Workflow

```
Sender(Client)                      Receiver(Server)
     ↓                                    ↓
  socket()                             socket()
     ↓                                    ↓
  connect() ------------+                 bind()
                        |                  ↓
                        |   SYN          listen()
            ACK         +--------------->   ↓
     +----------------------------------- accept()
     ↓              DATA/DATAACK            ↓
  sendto() <============================> recvfrom()
     ↓              FIN/FINACK              ↓
   close() ------------------------------>  ↓
     ↓    <---------------------------- close()
    done
```

## General

1. **socket() / bind() / listen()**

   Nothing very complicated. Just pass the parameters to UDP socket functions, providing wrapper functions.

2. **connect() / accept()**

   Unlike 3-way handshake in TCP, we implemented a 2-way handshake to setup a single direction channel.

3. **sendto() / recvfrom()**

   In sendto(), we implemented a sending sliding window that can be dynamically adjusted to control congestion.

   In recvfrom(), we implemented a fix-sized receiving window to help receive packets in correct order without loss.

4. **close()**

Unlike 4-way handshake in TCP, we implemented a 2-way handshake to close a single direction channel.

## Details & Discussion

**socket()**

Initiate variables in global socket state. Return result from OS socket().

**bind() / listen()**

Simple parameters checking. We know the end calling `listen()` is server. Hence we can setup signal handlers for server.

**connect() / accept()**

The setup process is divided into following parts.

1. Before connect()/accept(), client and server are in CLOSED state.
2. The client sends a SYN, with its state turned to SYN_SENT.
3. The server receives the SYN, responding with a SYNACK. Turn its state to SYN_RCVD.
4. The client receives the SYNACK and becomes ESTABLISHED.

Some details:

1. If SYN times out or gets lost, the timer on client side will resend a SYN for at most 5 times. Then it'll continue waiting for SYNACK.
2. If SYNACK times out or gets lost, the timer mentioned above will also resend SYN. The server will respond with SYNACK.

**sendto() / recvfrom()**

1. At this stage, the client will be in ESTABLISHED state, and the server will be in SYN_RCVD state. The client will begin to send data (sendto()), and the server will begin to receive data (recvfrom()).
2. Client will send DATA packets. At the same time, it keeps receiving/checking DATAACK to slide the sending window. Everytime it receives a new DATAACK, it will increase the size of window (switch a faster mode).
3. Server will receive packets. It may receive SYN, which means SYNs before this get lost. Then it will respond with SYNACK. When it receives DATA packets, turn its state to ESTABLISHED and start accumulate data. If it receives FIN, then respond with FINACK and turn its state to FIN_RCVD.

Some details:

1. If DATA gets lost and client doesn't receive DATAACK, the timer on client side will time out and resend a window of DATA. (Go-back-n) And switch to SLOW mode.
2. If DATAACK gets lost, the process is similar to above.
3. We also need to pay attention to parameter `len`.

   If `len` (send) is bigger than DATALEN, we need to cut sending data into several chunks and send them using sliding window.

On the recv side, we also need to reorganize the data payload to consecutive bytes then copy them to `buf`. At the same time we need to consider `len`.

**close()**

This is also kind of difficult to write the code. Because both of client and server can use this close() to close a connection.

1. If client first uses `close()`, it first sends a FIN and waits for FINACK. If it receives a FIN/FINACK, then close the sockfd.
2. Because server doesn't know whether the transmission is over or not, it need instructions from the other side. When `recvfrom()`, if it receives a FIN, it'll be FIN_RCVD. Then when it calls `close()` it first receives a FIN, respond with FINACK and turn to CLOSED.

Some details:

1. Because both side uses the same `close()` function, both will first send a FIN even it doesn't need to. Then it'll receives a packet. If it's FIN, then remote side want to close too. In this case we directly close sock. If it's FINACK, then remote received out FIN, we can also turn to CLOSED.

# Some tricky parts

Homework 1 is not a big software engineering project that needs a huge amount of codes, rather a very subtle and trivial code module which deals with lots of details and states.

The tricky, or we can say difficult part of this GBN-Socket module is:

1. We need to deal with transformations among multiple states. However, we can't simply do separate transformations in different functions. Instead, we need to detect/check different situations in many places and in different functions, which makes it hard to decouple codes into clear parts.
2. We need to handle timeouts in one single thread instead of multithreading, which makes implementation more complicated. We can't send and receive at the same time. Instead, we need some loop code structures to keep checking.
3. Two ends will both use the same set of functions. For example, client and server will both use close(), which requires us to consider different situations.
4. Obey the semantics of function names and take the right choice when dealing with some confusing problems.
5. Consider the parameter `len` in `recvfrom()` and `sendto()`.
6. Save the data temporarily for resending when timing out.