

# CSCI4430/ESTR4120 (Spring 2018)

## Assignment 2: Go-back-N

Due on March 15 (Thur), 23:59:59

### 1 Introduction

In this assignment, we will implement the Go-Back-N (GBN) protocol to support the reliable data transfer between a client and a server over UDP under an unreliable network connection.

### 2 Application Layer

The application that we consider is FTP. We have implemented two (very simple) FTP programs, namely `myftpserver` and `myftpclient`, in which `myftpclient` uploads a file to `myftpserver`. We make the following assumptions:

- We only support file uploads. We do not consider file downloads.
- The `myftpclient` program only uploads one file at a time and then quits.
- There is only one `myftpclient` connecting to `myftpserver` at a time.
- The file to be sent is located under the same working directory as the `myftpclient` program.
- `myftpserver` stores the file in a directory called `data`, which is located under same working directory as `myftpserver`. The file name should be the same as the original file name.

We will provide you two versions of the programs:

- The TCP versions of `myftpclient` and `myftpserver` that implement the FTP functionalities under TCP. You can use the programs to have an idea what the programs can do.
- The templates of `myftpclient.c` and `myftpserver.c` that implement our GBN protocol. Your implementation should be fully compatible with these two files.

### 3 GBN Design

#### 3.1 Structures

We define the structures for the GBN sender and the GBN receiver. Both structures should contain the socket descriptor plus other necessary fields that you define. Specifically, the structures are:

```
struct mygbn_sender {
    int sd;    // GBN sender socket
    // ... other member variables
};

struct mygbn_receiver {
    int sd;    // GBN receiver socket
    // ... other member variables
};
```

## 3.2 APIs

Our GBN protocol exports a set of APIs that are called by both `myftpclient` and `myftpserver` (see our programs). Your job is to provide implementation for the APIs. Specifically, `myftpclient` calls the following APIs:

- `void mygbn_init_sender(struct mygbn_sender* mygbn_sender, char* ip, int port, int N, int timeout)`: It initializes the sender socket and the related server IP address and port in `mygbn_sender`. It also specifies the parameter `N` for the GBN protocol and the retransmission timeout `timeout` in seconds.
- `int mygbn_send(struct mygbn_sender* mygbn_sender, unsigned char* buf, int len)`: It sends the data in `buf` of size `len` to the receiver. It returns the number of bytes that have been sent, or `-1` if there is any error.
- `void mygbn_close_sender(struct mygbn_sender* mygbn_sender)`: It terminates the sender connection, closes the sender socket, and releases all resources.

On the other hand, `myftpserver` calls the following APIs:

- `void mygbn_init_receiver(struct mygbn_receiver* mygbn_receiver, int port)`: It initializes the receiver socket and binds the port to the socket in `mygbn_receiver`.
- `int mygbn_recv(struct mygbn_receiver* mygbn_receiver, unsigned char* buf, int len)`: It receives the data in `buf` of size `len` to the receiver. It returns the size of packets that have been received, or `-1` if there is any error.
- `void mygbn_close_receiver(struct mygbn_receiver* mygbn_receiver)`: It closes the receiver socket and releases all resources.

Please note the following:

- Both `mygbn_send` and `mygbn_recv` should call only `sendto` and `recvfrom` for UDP transfers, respectively. Both `sendto` and `recvfrom` returns the number of bytes being sent or received.

## 3.3 GBN Packets

All packets of our GBN protocol are encapsulated under a protocol header defined as follows:

```
struct MYGBN_Packet {
    unsigned char protocol[3]; /* protocol string (3 bytes) 'gbn' */
    unsigned char type;        /* type (1 byte) */
    unsigned int seqNum;        /* sequence number (4 bytes) */
    unsigned int length;       /* length(header + payload) (4 bytes) */
    unsigned char payload[MAX_PAYLOAD_SIZE]; /* payload data */
} __attribute__((packed));
```

We define three types of packets: `DataPacket`, `AckPacket`, and `EndPacket`. Table 1 summarizes the definitions of their protocol fields.

In our GBN protocol, the sender sends a `DataPacket` with the payload to the receiver, which replies an `AckPacket` upon receiving the `DataPacket`. To terminate the data transfer connection, the sender sends an `EndPacket` to the receiver, which also replies an `AckPacket` upon receiving the `EndPacket`. Please note the following:

- To avoid fragmentation, we limit the `MAX_PAYLOAD_SIZE` as 512 bytes. To send a large payload, the sender needs to first partition the payload and then send multiple `DataPackets` to the receiver.
- You may assume that the sender initializes `seqNum` as one.
- When `mygbn_close_send` is called, the GBN sender sends an `EndPacket` to the receiver, which resets the cumulative acknowledgment number to zero.

<b>DataPacket</b>	protocol type seqNum length payload	“gbn” 0xA0 current sequence number total packet length (header length + payload length) application data
<b>AckPacket</b>	protocol type seqNum length	“gbn” 0xA1 cumulative acknowledgement number packet length (header length)
<b>EndPacket</b>	protocol type seqNum length	“gbn” 0xA2 current sequence number packet length (header length)

Table 1: GBN packet format.

### 3.4 Threads

Our GBN protocol leverages multi-threading (note that we still assume one client). On the sender side, we have at least two threads: (i) a thread for receiving AckPackets, and (ii) a thread for triggering the retransmissions upon timeouts. On the receiver side, we have at least a thread for receiving DataPackets and EndPackets. Please note the following:

- You are free to create as many threads as you want for performance optimization. However, all threads should be defined under the GBN structures (see Section 3.1).
- You need to use `pthread_cond_timedwait` to put a thread on sleep and wake up the thread upon timeouts. Do not use busy waiting to loop a thread. The TAs will talk more about how to use the function.

### 3.5 Timeouts

If the sender has not received the AckPacket for the oldest unacknowledged DataPacket after a timeout period, it retransmits up to  $N$  unacknowledged DataPackets. The timeout is configurable as a command-line parameter. If the sender receives the AckPacket for the oldest unacknowledged DataPacket and there are still additional unacknowledged DataPackets, it resets the timer to trigger a timeout event at time  $T + \tau$ , where  $T$  is the current clock time and  $\tau$  is the timeout value. As discussed in class, we have different ways to reset the timer, but we use this simple approach in this assignment.

### 3.6 Termination

When the sender calls `mygbn_close_send`, it sends an EndPacket to the receiver and waits for the AckPacket. If it does not receive anything after a timeout period, it retransmits the EndPacket. We allow the sender to retransmit by at most *three times*, and it will close the socket anyway and report an error message. If the receiver receives the EndPacket, it resets the cumulative acknowledgement number to zero to prepare for the next data transfer.

## 4 Network Setup

We create a lossy network that probabilistically drop packets. We provide a tool called *troller* that is installed on the receiver side. The troller intercepts all packets that are received from the network but not yet passed to the GBN protocol. It drops or reorders the intercepted packets with certain probabilities.

The troller is built on NFQUEUE. We will discuss NFQUEUE later in class. For now, you only need to follow the instructions to install the troller, without worrying how it is implemented. First, you need to install NFQUEUE on the VM that deploys the GBN receiver and `myftpsrvr`.

```
server> sudo apt-get update
server> sudo apt-get install libnetfilter-queue-dev
```

Then we set up NFQUEUE to intercept all UDP packets:

```
server> sudo iptables -t filter -F
server> sudo iptables -A INPUT -p udp -s $clientip -d $serverip \
        -j NFQUEUE --queue-num 0
server> sudo iptables -A OUTPUT -p udp -s $serverip -d $clientip \
        -j NFQUEUE --queue-num 0
```

We then install the troller to process the intercepted UDP packets.

```
server> sudo troller <drop_ratio> <reorder_ratio>
```

The parameter `drop_ratio` is a floating point number that specifies the probability that a packet is dropped, while the parameter `reorder_ratio` is also a floating number that specifies the probability that a packet is reordered.

## 5 Implementation Issues

The server uses the following command-line interface:

```
vm1> ./myfypserver <port number>
```

The client uses the following command-line interface:

```
vm2> ./myftpcient <server ip addr> <server port> <file> <N> <timeout>
```

Note that `file` is the input file, `N` is the parameter `N` in GBN, and `timeout` (in seconds) is the timeout period. Please note the following:

- The `myftpcient` program should terminate gracefully after it sends out a file successfully. On the other hand, the `myfypserver` can serve another data transfer session without restart.
- Our testing environment is Linux; more precisely, the Linux OS of your VMs.
- Your programs must be implemented in C or C++.
- Your programs must send/receive data under UDP.

## 6 Submission Guidelines

Please include all implementation under `mygbn.h` and `mygbn.c`. To make sure that you do not modify the original `myftp` code, you *must* submit a `Makefile` and both `mygbn.h` and `mygbn.c` **only**. Your `Makefile` should compile your code with the `myftp` code to generate an executable file. During the demo, we will integrate your submitted code with the original `myftp` code.

The deadline is March 15. We will arrange demo sessions to grade your assignments on March 16. Have fun! :)