CSCI4430/ESTR4120 (Spring 2019)

Assignment 2: Selective Repeat

Due on March 14 (Thur), 23:59:59

1 Introduction

In this assignment, we will implement the Selective Repeat (SR) 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 SR protocol. Your implementation should be fully compatible with these two files.

3 SR Design

3.1 Structures

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

```
struct mysr_sender {
   int sd;    // SR sender socket
   // ... other member variables
};

struct mysr_receiver {
   int sd;    // SR receiver socket
   // ... other member variables
};
```

3.2 APIs

Our SR 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 mysr_init_sender(struct mysr_sender* mysr_sender, char* ip, int port, int N, int timeout)`: It initializes the sender socket and the related server IP address and port in `mysr_sender`. It also specifies the window size `N` for the SR protocol and the retransmission timeout `timeout` in seconds.
- `int mysr_send(struct mysr_sender* mysr_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 mysr_close_sender(struct mysr_sender* mysr_sender)`: It terminates the sender connection, closes the sender socket, and releases all resources.

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

- `void mysr_init_receiver(struct mysr_receiver* mysr_receiver, int port)`: It initializes the receiver socket and binds the port to the socket in `mysr_receiver`.
- `int mysr_recv(struct mysr_receiver* mysr_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 mysr_close_receiver(struct mysr_receiver* mysr_receiver)`: It closes the receiver socket and releases all resources.

Please note the following:

- Both `mysr_send` and `mysr_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 SR Packets

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

```
struct MYSR_Packet {
  unsigned char protocol[2]; /* protocol string (2 bytes) ''sr'' */
  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 3.3 summarizes the definitions of their protocol fields.

In our SR 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 `mysr_close_send` is called, the SR sender sends an EndPacket to the receiver, which resets its state.

DataPacket	`protocol`	"sr"
	`type`	0xA0
	`seqNum`	current sequence number
	`length`	total packet length (header length + payload length)
	`payload`	application data
AckPacket	`protocol`	"sr"
	`type`	0xA1
	`seqNum`	seqNum of the acknowledged DataPacket
	`length`	packet length (header length)
EndPacket	`protocol`	"sr"
	`type`	0xA2
	`seqNum`	current sequence number
	`length`	packet length (header length)

Table 1: SR packet format.

3.4 Threads

Our SR 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 needed. However, all threads should be defined under the SR 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 an unacknowledged DataPacket after a timeout period, it retransmits that DataPacket. The timeout is configurable as a command-line parameter.

3.6 Termination

When the sender calls `mysr_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 everything it needs 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 SR 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 SR receiver and `myftpserver`.

```
server> sudo -E apt-get update
server> sudo -E 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> ./myftpclient <server ip addr> <server port> <file> <N> <timeout>
```

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

- The `myftpclient` program should terminate gracefully after it sends out a file successfully. On the other hand, the `myftpserver` can serve another data transfer session without restart.
- Our testing environment is Linux only; 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 `mysr.h` and `mysr.c`. To make sure that you do not modify the original myftp code, you *must* submit a Makefile and both `mysr.h` and `mysr.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 14. We will arrange demo sessions to grade your assignments on March 15. Have fun! :)