Lab 6
Stop and Wait for Unreliable Channel with Packet Loss

COEN 146L Spring 2021

Reminders

- All the labs are due at 11:59 PM on the next Monday after the lab has been assigned
- TA office hours:
  - Rachael (rfreitag@scu.edu)
    - Monday: 12 - 1 pm
    - Tuesday: 2:30 - 4 pm
    - Thursday: 11 am - 12:30 pm
  - Jesse (jschen@scu.edu)
    - Monday: 1 - 2 pm
    - Tuesday, Friday: 5 - 6 pm
  - Or by appointment

Lab 6 Objectives

- Concept: Transfer data between application using UDP/IP sockets
- Concept: Implement a reliable stop&wait protocol(rdt3.0) on top of UDP
  - accomplish reliability through checksums, sequence numbers, and timeout retransmissions
- Programming: Write C sender/receiver applications to **reliably** transfer a file using UDP/IP sockets.
  - will be handling dropped packets

Concepts

- File I/O: Lab 2, 4, 5
- Socket Programming: Lab 4, 5
- RDT protocol: Lab 5
  - rdt3.0 = rdt2.2 + timeouts/retransmissions
- socket timeouts: new!
  - accomplished through select()
  - also need to simulate dropped packets

select()

- allows a program to monitor multiple file descriptors to wait for one to become ready for I/O operation(read/write/"exceptional conditions")
  - high level: takes in a set of sockets, and returns when a socket is ready to read from, OR the timer runs out before any socket is ready
- `int` **`select(int`** *`nfds`*`,` **`fd_set`** `*`*`readfds`*`,` **`fd_set`** `*`*`writefds`*`,` **`fd_set`** `*`*`exceptfds`*`,` **`struct timeval`** `*`*`timeout`***`);`**
  - `int nfds` - upper limit of file descriptors to be checked (in code, set to **`sockfd+1`**)
  - `fd_set *readfds, *writefds, *exceptfds` - sets of file descriptors to be checked
    - only need to use `readfds` for this lab, `writefds` and `exceptfds` can be set to `NULL`
    - must be initialized before using
  - `struct timeval *timeout` - structure to hold the timeout length
    - how long to monitor the file descriptors/sockets for

select() prep

1. socket and sockaddr_in prep as usual
2. set up **how long to wait for** using a timeval struct
   a.
```
struct timeval tv;
tv.tv_sec = time_out; // seconds
tv.tv_usec = 0; // microseconds
```
3. set up **which file descriptors/sockets to monitor** using fd_sets
   a.
```
fd_set readfds; // creates the fd_set
FD_ZERO(&readfds); // clears the fd_set
FD_SET(server_fd, &readfds); // adds server_fd to the fd_set
```
4. call select() using initialized fd_set and timeval
   a.
```
select(server_fd+1, &readfds, NULL, NULL, &tv);
```
   b. then, we check the return value and react accordingly

select() return values

- non-error: returns the number of ready file descriptors from the fd_sets it monitored
  - **0 means no file descriptors were ready before the timeout**
  - **important**: the fd_sets are modified in the course of a select() call, and will need to be reinitialized before select() can be called again.
  - each fd_set now contains the file descriptors/sockets that are ready
- error: -1

- check the return value and then write your program to behave accordingly

example

```c
// ./tcp_client <server_ip> <port>
int main(int argc, char* argv[]) {
    char* server_ip = argv[1]; // server ip
    int port = atoi(argv[2]);  // port number
    char* message = "hello world!";


    // (1) opens a socket
    int socket_fd = socket(AF_INET, SOCK_DGRAM, 0);

    // initializes a sockaddr_in structure to connect to
    struct sockaddr_in address;
    address.sin_family = AF_INET; // IPv4
    address.sin_addr.s_addr = inet_addr(server_ip); // converts ip from char* to binary
    address.sin_port = htons(port); // converts from host to network byte order

    // (2) send message to server
    int bytes_sent = sendto(socket_fd, message, strlen(message)+1, 0, (struct sockaddr*)&address, sizeof(
        address));
    printf("sent %i bytes!\n", bytes_sent);

    // (3) cleanup
    close(socket_fd);
}
```

doesn't call select()

(this is the same file as lab4 example)

```c
// ./select_server <port> <timeout>
int main(int argc, char* argv[]) {
    int port = atoi(argv[1]);
    int time_out = atoi(argv[2]);
    void* buf = malloc(BUF_SIZE);

    // (1) open a socket
    int server_fd = socket(AF_INET, SOCK_DGRAM, 0);

    // initializes a sockaddr_in structure to listen on
    struct sockaddr_in address, client_address;
    address.sin_family = AF_INET; // IPv4
    address.sin_addr.s_addr = htonl(INADDR_ANY); // binds to all available interfaces
    address.sin_port = htons(port); // converts from host to network byte order
    int addrlen = sizeof(address);

    // (2) bind address so client knows who to send a message to
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));

    // (3) wait for a message for time_out seconds using select()
    // use a struct timeval to represent how long to wait for
    struct timeval tv;
    tv.tv_sec = time_out; // seconds
    tv.tv_usec = 0; // microseconds

    // set up the fd_set
    fd_set readfds; // creates the fd_set
    FD_ZERO(&readfds); // clears the fd_set
    FD_SET(server_fd, &readfds); // adds server_fd to the fd_set

    // monitor file descriptors/sockets in readfds to see if they are ready for reading
    // "ready for reading" == a read() call on that socket will not block
    int select_return = select(server_fd+1, &readfds, NULL, NULL, &tv);

    if (select_return > 0) { // one or more socket is ready and can be read from
        printf("***%i SOCKET(S) READY**\n", select_return);
        printf("is server_fd in readfds? %i\n", FD_ISSET(server_fd, &readfds));
        int byte_recvd = recvfrom(server_fd, buf, BUF_SIZE, 0, (struct sockaddr*)&client_address, &addrlen);
        printf("received message: %s\n", (char*)buf);
    }
    else { // no sockets were ready after the timeout
        printf("**TIMEOUT**\n");
        printf("is server_fd in readfds? %i\n", FD_ISSET(server_fd, &readfds));
    }

    // (4) cleanup
    close(server_fd);
    free(buf);
}
```

calls select()

rdt3.0

- pretty much the same as rdt2.2, but can handle dropped packets through retransmission
  - in rdt2.2, if a packet is dropped, the program will hang forever
  - rdt3.0 will retransmit the last sent packet if an ACK is taking too long to arrive

Steps

1. Implement rdt3.0 over UDP
    a. starter code is on the assignment page
        i. Lab6clientBarebone.c - sender
        ii. Lab6serverBarebone.c - receiver
        iii. lab6.h - Packet/Header structs
2. Modify Step 1 to randomly simulate dropped packets
    a. use rand()
    b. no need to simulate checksum and sequence number errors for this lab
        i. but you can tho

Notes

- do not use strlen() on data buffers!
  - by far the most common mistake on lab 5
  - your code should be able to work on binary data
- save the output of any read() call
  - read does not always fill up the entire buffer
- if you are unsure what the middle two arguments of fread() are/do, just use read()

Questions?