# CS4220 Project 1: Socket-based ARQ Protocol Programming (due date: February 12, 2017)

11. A file server in TCP

The source code of both client and server side are:

```
/* This page contains the client program. The following one contains the
 * server program. Once the server has been compiled and started, clients
 * anywhere on the Internet can send commands (file names) to the server.
 * The server responds by opening and returning the entire file requested.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345             /* arbitrary, but client and server
must agree */
#define BUF_SIZE 4096                 /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];                 /* buffer for incoming file */
  struct hostent *h;                  /* info about server */
  struct sockaddr_in channel;         /* holds IP address */

  if (argc != 3) fatal("Usage: client server-name file-name");
  h = gethostbyname(argv[1]);         /* look up host's IP address */
  if (!h) fatal("gethostbyname failed");

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s < 0) fatal("socket");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
  channel.sin_port= htons(SERVER_PORT);
  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) fatal("connect failed");

  /* Connection is now established. Send file name including 0 byte at end.
*/
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
        bytes = read(s, buf, BUF_SIZE);       /* read from socket */
        if (bytes <= 0) exit(0);      /* check for end of file */
        write(1, buf, bytes);         /* write to standard output */
  }
}
```

```
fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345             /* arbitrary, but client and server
must agree */
#define BUF_SIZE 4096                 /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
  int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];                 /* buffer for outgoing file */
  struct sockaddr_in channel;         /* hold's IP address */

  /* Build address structure to bind to socket. */
  memset(&channel, 0, sizeof(channel));       /* zero channel */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Passive open. Wait for connection. */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
  if (s < 0) fatal("socket failed");
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b < 0) fatal("bind failed");

  l = listen(s, QUEUE_SIZE);          /* specify queue size */
  if (l < 0) fatal("listen failed");

  /* Socket is now set up and bound. Wait for connection and process it. */
  while (1) {
        sa = accept(s, 0, 0);         /* block for connection request */
        if (sa < 0) fatal("accept failed");

        read(sa, buf, BUF_SIZE);      /* read file name from socket */

        /* Get and return the file. */
        fd = open(buf, O_RDONLY);     /* open the file to be sent back */
        if (fd < 0) fatal("open failed");

        while (1) {
                bytes = read(fd, buf, BUF_SIZE); /* read from file */
                if (bytes <= 0) break;         /* check for end of file */
                write(sa, buf, bytes);         /* write bytes to socket */
        }
        close(fd);                             /* close file */
        close(sa);                             /* close connection */
```

```
  }
}

fatal(char *string)
{
  printf("%s", string);
  exit(1);
}
```

Current client code displays the (text) file on the standard output. Modify client side so that a binary file, transferred from the server, will be saved as a file in the local directory. Make both sides running on Unix computers (by remote access). Note that currently these servers are accessible: blanca, eas-gcc, shavano, windom, crestone, redcloud, and sanluis. Since those Unix machines are sharing the same file system, you may create a new directory to save the received file at the client side.

Server side: TCP_server
Client side: TCP_client server_hostname filename

TCP_client is the executable program. server_hostname is the hostname of the machine you use as the server. filename is the file that client wants to read from the server. We assume that the server has the requested file (you can store several PDF files there). Use a PDF file to test your programs. You may use binary I/O in the file operations. You may also want to save the transmitted file in a different directory due to the use of a shared file system in our Unix computers (by remote access).

You should be able to open the test PDF file on the client machine after the TCP transmission is completed. Note that you may use 2XYZ as the port number in your program, in which XYZ are the last 3 digits of your student ID. Doing so would avoid that multiple programs/processes use the same port for communication.

3.2. A file server in UDP (with flow control)

Modify the source code of 3.1 using UDP sockets (instead of TCP sockets). Make both sides running on Unix computers (by remote access). See the following programs as reference:

```
// UDP Echo Client

#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64
```

```
long delay(struct timeval t1, struct timeval t2)
{
        long d;
        d = (t2.tv_sec - t1.tv_sec) * 1000;
        d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
        return(d);
}
int main(int argc, char **argv)
{
        int     data_size = DEFLEN, port = SERVER_UDP_PORT;
        int     i, j, sd, server_len;
        char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
        struct  hostent         *hp;
        struct  sockaddr_in     server;
        struct  timeval         start, end;
        unsigned long address;

        pname = argv[0];
        argc--;
        argv++;
        if (argc > 0 && (strcmp(*argv, "-s") == 0)) {
                if (--argc > 0 && (data_size = atoi(*++argv))) {
                        argc--;
                        argv++;
                }
                else {
                        fprintf(stderr,
                        "Usage: %s [-s data_size] host [port]\n", pname);
                        exit(1);
                }
        }
        if (argc > 0) {
                host = *argv;
                if (--argc > 0)
                        port = atoi(*++argv);
        }


        else {
                fprintf(stderr,
                "Usage: %s [-s data_size] host [port]\n", pname);
                exit(1);
        }

        if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }

        bzero((char *)&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "Can't get server's IP address\n");
                exit(1);
        }
```

```
        bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);

        if (data_size > MAXLEN) {
                fprintf(stderr, "Data is too big\n");
                exit(1);
        }
        for (i = 0; i < data_size; i++) {
                j = (i < 26) ? i : i % 26;
                sbuf[i] = 'a' + j;
        } // construct data to send to the server
        gettimeofday(&start, NULL); /* start delay measurement */
        server_len = sizeof(server);
        if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)
                &server, server_len) == -1) {
                fprintf(stderr, "sendto error\n");
                exit(1);
        }
        if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)
                &server, &server_len) < 0) {
                fprintf(stderr, "recvfrom error\n");
                exit(1);
        }
        gettimeofday(&end, NULL); /* end delay measurement */
        if (strncmp(sbuf, rbuf, data_size) != 0)
                printf("Data is corrupted\n");
        close(sd);
        return(0);
}
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT               5000
#define MAXLEN         4096

int main(int argc, char **argv)
{
        int     sd, client_len, port, n;
        char    buf[MAXLEN];
        struct  sockaddr_in    server, client;

        switch(argc) {
        case 1:
                port = SERVER_UDP_PORT;
                break;
        case 2:
                port = atoi(argv[1]);
                break;
        default:
                fprintf(stderr, "Usage: %s [port]\n", argv[0]);
                exit(1);
        }

        /* Create a datagram socket */
        if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

```
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }


        /* Bind an address to the socket */
        bzero((char *)&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(sd, (struct sockaddr *)&server,
        sizeof(server)) == -1) {
                fprintf(stderr, "Can't bind name to socket\n");
                exit(1);
        }

        while (1) {
                client_len = sizeof(client);
                if ((n = recvfrom(sd, buf, MAXLEN, 0,
                (struct sockaddr *)&client, &client_len)) < 0) {
                        fprintf(stderr, "Can't receive datagram\n");
                        exit(1);
                }

                if (sendto(sd, buf, n, 0,
                (struct sockaddr *)&client, client_len) != n) {
                        fprintf(stderr, "Can't send datagram\n");
                        exit(1);
                }
        }
        close(sd);
        return(0);
}
```

Then, enhance the reliability of your code of 3.2(a) by implementing two ARQ algorithms; stop-and-wait (SW), and one of go back N (GBN) and selective repeat (SR). Refer to the book for sequencing, ACKs, and retransmission mechanisms. The communication should be reliable, assuring that the client got whole file correctly. Since UDP indeed is indeed quite reliable for communications within a LAN, use a random generator/function to "purposely" drop X% of data blocks BEFORE they are delivered from the server side. Each UDP segment should be less than 4KB, so that you can have enough segments to experience a few segment losses.

Server side: UDP_server loss_probability protocol_type Client side: UDP_client server_hostname file_name protocol_type Parameter loss_probability is the dropping probability (an integer) as a command-line input of the server code. Parameter protocol_type is the ARQ type implemented (1 for SW, 2 for GBN, 3 for SR). server_hostname is the hostname of the machine you use as the server (e.g., use WINDOM server). filename is the file that client wants to read from the server. Clearly, we assume that the server has the requested file. Use a PDF file to test your programs. You should be able to open the test PDF file on the client machine. You may want to save the transmitted file in a different directory due to the use of a shared file system in our Unix/Unix computers. Again, you may use 2XYZ as the port number in your

program, in which XYZ are the last 3 digits of your student ID. Doing so would avoid that multiple programs/processes use the same port for communication.

## What to turn in:

For this project you will email me at ftorres@uccs.edu one ZIP file (in the name of P2_Yourlastnames) of all source code, together with a technical report specifying 1) how to compile and execute your program; 2) the self-testing results of your program and any important feature you want to specify (if your program does not work, please notify it clearly). Your program should print the trace of flow control for the lost frames (i.e., Seq#, ACK#, Retransmission#) into an output text file. Please also have the project report in a hardcopy at the project submission time. Make sure your code can work in Unix machines (by remote access). We may check your project during the class time by remotely accessing servers.

You may search the Internet for more helpful information on socket-based network programming.