Dan Kass
Networking II CE-4960
4/17/2014
Lab 4
Super-Stateless File Transfer Protocol

User Instructions:

The Program uses command line arguments with the function getopt.

```
-a <address>     Required address to server.
-f <filename>    Required for requesting file or file size
-l               Flag to request file listing
-p <port number> Optional defaults to 22222
-r               Flag to request a file
-s <size>        Optional defaults to 1400 Bytes
-x               Flag to request file size
```

Lab Experience:

My lab experience for this lab wasn't to bad. I started off with your UDP Echo Client and adapted it for the functionality of this lab. It was really interesting learning about getopt for getting and organizing the command line arguments. It was a bit complicated to start off with it, but once it was working it made managing the command line arguments really quite simple and easy to manage. Then working with the struct took a little bit to remember how to do again but I was able to get that working. It was fun being able to download files using your own program. The attackgoat.gif was pretty funny too!

IP fragmentation Observations:

IP fragmentation happens when a datagram gets broken up into smaller parts to send them quicker, but if one of those packets that got fragmented were to get lost then the whole packet would have to be sent again not just the small part of the fragmented packet that was lost.

Source Code for Lab 4:

```
/*
 ============================================================================
 Name        : lab4.c
 Author      : Dan Kass
 Class       : Networking II (CE 4960)
 Description : SSFTP (Super Stateless File Transfer Protocol) Client
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <inttypes.h>

#define MAX_MESSAGE 65467
#define HEADER_LENGTH 40

#define REQUEST_FILE 0
#define RESPONSE_FILE 1
#define REQUEST_SIZE 2
#define RESPONSE_SIZE 3
#define REQUEST_LIST 4
#define RESPONSE_LIST 5
#define ERROR_FILE 6
#define ERROR_INVALID 7

char *UDP(char *buffer, unsigned short port, char *address);

typedef struct __attribute((packed)) ssftp_packet ssftp_packet;
struct ssftp_packet {
      uint8_t operation;
      uint8_t flags;
      uint16_t length;
      uint32_t offset;
      char filename[32];
      char data[MAX_MESSAGE];
};

int main(int argc, char **argv) {

      //Variables.
      ssftp_packet ssftp_send;
      ssftp_packet ssftp_receive;

      uint8_t operation = 0;
      uint8_t flags = 0;
      uint16_t length = 1400;
      uint32_t offset = 0;
      char *file = NULL;
      char *address = NULL;
```

```
unsigned short port = 22222;
float size;
char buffer[MAX_MESSAGE + HEADER_LENGTH];

FILE *fp;

char *help = "-a <address>     (required)\n"
                   "-f <filename>    (required only if -r or -x)\n"
                   "-l               --Operation File Listing\n"
                   "-p <port number> (Optional defaults to 22222)\n"
                   "-r               --Operation Request File\n"
                   "-s <size>        (Optional defaults to 1400 Bytes)\n"
                   "-x               --Operation Request File Size\n";

//make sure there is enough arguments.
if(argc < 2){
      printf("%s",help);
      exit(1);
}

int c;
while((c = getopt(argc,argv, "a:f:hlp:rs:x")) != -1){
      switch(c) {
      case 'a':
            //gets host address
            address = optarg;
            break;
      case 'f':
            //request file name
            file = optarg;
            break;
      case 'h':
            printf("%s",help);
            exit(0);
            break;
      case 'l':
            //request file listing
            operation = REQUEST_LIST;
            break;
      case 'p':
            //gets port number
            port = atoi(optarg);
            break;
      case 'r':
            //request file
            operation = REQUEST_FILE;
            break;
      case 's':
            //gets file size block
            length = atoi(optarg);
            break;
      case 'x':
            //request file size
            operation = REQUEST_SIZE;
            break;
      default:
            break;
      }
```

```
}

if(address == NULL){
      printf("Please Enter an Address");
      exit(1);
}

if(length > MAX_MESSAGE){
      length = MAX_MESSAGE;
}

if((operation == REQUEST_FILE) || (operation == REQUEST_SIZE)){
      if(file == NULL){
            printf("Please Enter a Filename");
            exit(1);
      }
}

switch (operation) {
case REQUEST_FILE:
      //so first we need to determine the file size
      // init struct and copy to buffer
      ssftp_send.operation = REQUEST_SIZE;
      ssftp_send.flags = flags;
      ssftp_send.length = htons(length);
      ssftp_send.offset = htonl(offset);
      memcpy(ssftp_send.filename, file, strlen(file));

      //copy to buffer
      memcpy(buffer, &ssftp_send, HEADER_LENGTH);

      memcpy(&ssftp_receive, UDP(buffer, port, address),
                  MAX_MESSAGE + HEADER_LENGTH);

      if (ssftp_receive.operation == RESPONSE_SIZE) {
            size = htonl(ssftp_receive.offset);

      } else {
            printf("There was an error:%d\n", ssftp_receive.operation);
            exit(1);
      }
      //then we have to determine how many times we need to
      //request to the server to get the whole file.
      int numberOfPackets = size / length;


      //loop through the calculated amount to get the whole file
      //and save it to the folder it comes in a binary format
      int i;
      remove(ssftp_send.filename);
      //printf("Need %d\n",numberOfPackets);
      printf("Writing to file..");
      for (i = 0; i < numberOfPackets+1; i++) {
            offset = (i * length);

            ssftp_send.operation = REQUEST_FILE;
            ssftp_send.flags = flags;
            ssftp_send.length = htons(length);
```

```
                ssftp_send.offset = htonl(offset);

                memcpy(ssftp_send.filename, file, strlen(file));
                //copy to buffer

                memcpy(buffer, &ssftp_send, HEADER_LENGTH);
                memcpy(&ssftp_receive, UDP(buffer, port, address),
                        MAX_MESSAGE + HEADER_LENGTH);

                fp=fopen(ssftp_receive.filename, "a+");
                fwrite(ssftp_receive.data, 1, htons(ssftp_receive.length), fp);
                fclose(fp);
            }
                printf("..done\n");
            break;

        case REQUEST_SIZE:
            // init struct and copy to buffer
            ssftp_send.operation = operation;
            ssftp_send.flags = flags;
            ssftp_send.length = htons(length);
            ssftp_send.offset = htonl(offset);
            memcpy(ssftp_send.filename, file, strlen(file));

            //copy to buffer
            memcpy(buffer, &ssftp_send, HEADER_LENGTH);

            //using UDP send and get the response from the server
            memcpy(&ssftp_receive, UDP(buffer, port, address),
                    MAX_MESSAGE + HEADER_LENGTH);

            if (ssftp_receive.operation == RESPONSE_SIZE) {
                size = htonl(ssftp_receive.offset);
                size = size / 1000;
                printf("The size of %s is %.2f kB\n", ssftp_receive.filename,
                        size);
            } else {
                printf("There was an error:%d\n", ssftp_receive.operation);
            }
            break;

        case REQUEST_LIST:
            // init struct and copy to buffer
            ssftp_send.operation = operation;
            ssftp_send.flags = flags;



            //copy to buffer
            memcpy(buffer, &ssftp_send, HEADER_LENGTH);

            memcpy(&ssftp_receive, UDP(buffer, port, address),
                    MAX_MESSAGE + HEADER_LENGTH);

            if (ssftp_receive.operation == RESPONSE_LIST) {
                printf("File Listing from %s\n%s", address, ssftp_receive.data);
            } else {
                printf("There was an error:%d\n", ssftp_receive.operation);
            }

            break;
      }

      exit(0);
}

/**************************************************
 *  Function:    UPD
 *  Parameters:  ssftp_packet to send,
 *                 int port number
 *                 char *address to send to
 *  Returns:     ssftp_packet that was returned.
 *  Description: This function sends and receives
 *    the packet from the socket
 **************************************************/
char *UDP(char buffer[MAX_MESSAGE + HEADER_LENGTH], unsigned short port,
            char *address) {

      // locals
      int sock;
      struct sockaddr_in server;
      struct hostent *hp;

      // create socket
      // IP protocol family (PF_INET)
      // UDP (SOCK_DGRAM)

      if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
            perror("Error creating socket");
            exit(1);
      }

      // UDP  Using UDP we don't need to call bind unless we
      // want to specify a "source" port number.  We really
      // do not care - server will reply to whatever port we
      // are given

      // Make a sockaddr of the server
      // address family is IP  (AF_INET)
      // server IP address is found by calling gethostbyname with the
      // name of the server (entered on the command line)
      // note, if an IP address is provided, that is OK too

      server.sin_family = AF_INET;

      if ((hp = gethostbyname(address)) == 0) {
            perror("Invalid or unknown host");
            exit(1);
      }

      // copy IP address into address structure
      memcpy(&server.sin_addr.s_addr, hp->h_addr, hp->h_length);

      // establish the server port number - we must use network byte order!

      server.sin_port = htons(port);
```

```
    int size_to_send;
    int size_sent;
    int size_echoed;

    // how big?
    size_to_send = HEADER_LENGTH;

    // send to server
    size_sent = sendto(sock, buffer, size_to_send, 0,
                (struct sockaddr*) &server, sizeof(server));

    if (size_sent < 0) {
        perror("Error sending data");
        exit(1);
    }

    // clear buffer
    memset(buffer, 0, MAX_MESSAGE);

    // Wait for a reply - from anybody

    // Hold-on, what if we missed the reply, or it never comes.
    // we would normally just do this:
    //    size_echoed = recvfrom(sock, buffer, MAX_MESSAGE,0 , NULL, NULL);
    // and be on our way.  However, recvfrom is blocking.
    // Need to check to make sure there is something to read before we
    // do the deed.  Of course, in true Unix fashion, this is not easy.

    // the timeout
    struct timeval tv = { 5, 0 }; // 5 second, 0 milliseconds

    // file descriptor set
    fd_set socketReadSet;
    FD_ZERO(&socketReadSet);
    FD_SET(sock, &socketReadSet);

    if (select(sock + 1, &socketReadSet, 0, 0, &tv) < 0) {
        perror("Error on select");
        exit(0);
    }

    if (FD_ISSET(sock,&socketReadSet)) {
        size_echoed = recvfrom(sock, buffer, MAX_MESSAGE, 0, NULL, NULL);

        if (size_echoed < 0) {
            perror("Error receiving");
            exit(1);
        }

    } else {
        printf("Timeout - Server is not listening, or packet was dropped.\n");
        exit(1);
    }

    return buffer;
}
```