File Transfer Client and Server using Vigenere Cipher

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on October 15, 2010 November 12, 2010 About Programming / Algorithms / ANSI/POSIX C

[Disclaimer: This code is provided for educational purposes, you are responsible of how you use this code and this code is provided 'as is'. Meaning that I do not take responsibility of any harm that this code may or not produce]

The following is an example of a client and server that let you transfer a file from the client to the server by encrypting and then decrypting the packages using Vigenere Cipher.

Test file content example:

Note: This program is designed only to send files that are alphabetic in lower case, without spaces, and only one line.

```
abcdefghijklmnopqrstuvwxyzaabcdefghijklmnopqrstuvwxyzaabcdefghijklmnopqrstuvwxyza
```

Makefile:

As a good policy, always add your name, short description, and any other information that could help a user that doesn't know as you the content of your code and its behaviour.

Note: In the server, there must be a subdirectory named cli_serv in which ssh and scp can access after loading.

```
#########################################################################
# Author: Alejandro G. Carlstein Ramos Mejia
# Description: Client/Server File Transfer using Vergene Cipher
#
#
# make <= compile all files
# make build <= compile all files
# make all <= clean and compile all files
# make ssh <= Connect to server and go to folder cli_serv
# make upload <= Upload files on SRC_FILES list to server
# make uploadssh <= Upload files on SRC_FILES list to server and connect to server
# make submit <= Tar files on SRC_FILES list
# make zipsubmit <= Tar files on SRC_FILES list and gzip them
# make clean <= Clean all executable and *.o files
# make debug_server <= Debug server using dbg
# make debug_client <= Debug client using dbg
# make ldebug_server <= Debug for leaks on server
# make ldebug_client <= Debug for leaks on client
##########################################################################
```

```
##########################################################################
# Variables
##########################################################################

# PROJECT is the name used when preparing for sumit/
# The tar and/or zip file will be using this name
PROJECT = server_client_vergene

# SRC_FILES are list of files in the project.
# This list is used when uploading files to the server.
# Also, this list is used when tar/zipped for submit
SRC_FILES = \
 cli.c \
 serv.c\
 default.h \
 default.c \
 text.txt\
 Makefile \
 README

# OpenSSH SSH client (remote login program)
SSH = ssh

# secure copy (remote file copy program)
# SCP is used for uploading files to the server in secure mode
SCP = scp

# FOLDER_SERVER is the folder that SSH and SCP will try to access after login
FOLDER_SERVER = server_client_vergene

# SSH_SERVER is the hostname of the server
SSH_SERVER = bingsuns2.cc.binghamton.edu

# SSH_OPTION
# -t opens a pseudo-tty with in the current session.
# This flag is required to execute the commands on SSH_CD_FOLDER
SSH_OPTION = -t

# SSH_CD_FOLDER executes cd
# Then change the prompt to show the number of bash in the server
SSH_CD_FOLDER = 'cd $(FOLDER_SERVER); bash; echo $PS1'

# CC indicates which compiler is going to be used
CC = gcc

# Files required for compiling the server
CODE_SERVER_FILE = serv.c default.c

# Files required for compiling the client
CODE_CLIENT_FILE = cli.c default.c

# Name of the executable file for the server
```

```
EXEC_SERVER_FILE = serv

# Name for the executable file for the client
EXEC_CLIENT_FILE = cli

# Flags for the compiler
# -g indicate to provide debuggin information
# -Wall activates the warnings.
# -lm indicate the compiler to add basic mathematics libraries
CFLAGS = -g -Wall -lm

# COMPILE is the combination of the compiler with the flags
COMPILE = $(CC) $(CFLAGS)

# MFLAGS are flags that require to be added at the end
MFLAGS =

# USERNAME is used later when required to do an upload followed with ssh
USERNAME =

# Detect if the computer is SunOS to add flags needed for compiling
UNAME := $(shell uname)
ifeq ($(UNAME), SunOS)
 MFLAGS := -lsocket -lnsl
endif

###########################################################################
# 'make' options
###########################################################################

# Clean all files and compile client and server
all: clean compile_server compile_client

# Just build server and client
build: compile_server compile_client

# Compile server
compile_server: $(CODE_SERVER_FILE)
 $(CC) $(CFLAGS) -o $(EXEC_SERVER_FILE) $(CODE_SERVER_FILE) $(MFLAGS)

# Compile client
compile_client: $(CODE_CLIENT_FILE)
 $(CC) $(CFLAGS) -o $(EXEC_CLIENT_FILE) $(CODE_CLIENT_FILE) $(MFLAGS)

# Debug server
debug_server:
 gdb $(EXEC_SERVER_FILE)

# Debug client
debug_client:
 gdb $(EXEC_CLIENT_FILE)

# Leak debug server
```

```
ldebug_server:
 valgrind --leak-check=full --show-reachable=yes -v $(EXEC_SERVER_FILE)

# Leak debug client
ldebug_client:
 valgrind --leak-check=full --show-reachable=yes -v $(EXEC_CLIENT_FILE)

# Upload files and connect to server
uploadssh:
 @echo -n 'Upload and connect to $(SSH_SERVER)- USERNAME: ';\
 read user_name;\
 $(SCP) $(SRC_FILES) $$user_name@$(SSH_SERVER):./$(FOLDER_SERVER);\
 $(SSH) $(SSH_OPTION) $$user_name@$(SSH_SERVER) $(SSH_CD_FOLDER);\

# Upload files to server
upload:
 @echo -n 'Upload to $(SSH_SERVER)- USERNAME: ';\
 read user_name;\
 $(SCP) $(SRC_FILES) $$user_name@$(SSH_SERVER):./$(FOLDER_SERVER);\

# Connect to server
ssh:
 @echo -n 'Connect to $(SSH_SERVER)- USERNAME: ';\
 read user_name;\
 $(SSH) $(SSH_OPTION) $$user_name@$(SSH_SERVER) $(SSH_CD_FOLDER);\

# Makes a archive containing all the project source files for submission.
submit: $(SRC_FILES)
 tar -cvf $(PROJECT).tar $(SRC_FILES)

# Makes a archive containing all the project source files for submission
# and zip them
zipsubmit: $(SRC_FILES)
 tar cvfz $(PROJECT).tar.gz $(SRC_FILES)

# Clean files
.PHONY: clean
clean:
 rm -f *~ *.o $(EXEC_SERVER_FILE) $(EXEC_CLIENT_FILE)
```

Default file is required for both the server and the client. It holds the libraries and definitions shared by both programs.

Default.h file:

```
/**
 * Author: Alejandro G. Carlstein
 * Description: File Transfer Client/Server
 */

#ifndef ACARLS_DEFAULT_H
#define ACARLS_DEFAULT_H
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include <dirent.h>

/* Required for linux */
#include <string.h>

/* Required for SunOS */
#include <strings.h>

#include <ctype.h>

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

#define DEFAULT_PORT      8414
#define DATA_PACKET_SIZE    1024
#define NUM_BYTES       8
#define ENCRYPTION_KEY      'security'
#define CHAR_KEY_A      97
#define CHAR_PLAIN_A     97
#define CHAR_CIPHER_A     65

#define DBG_LV0 0
#define DBG_LV1 0
#define DBG_LV2 0
#define DBG_LV3 0

#define TRUE         1
#define FALSE         0

#define QUIT_TRUE       0
#define QUIT_FALSE      1

#define CODE_FAIL       '0FAIL'
#define CODE_OK        '1OK'

#define CODE_HELLO      '100HELLO'
#define CODE_WELCOME      '101WELCOME'
#define CODE_REQ_SERVER_NAME  '102REQ_SERVER_NAME'

#define CODE_MSG        '200MSG'

#define CODE_DATA       '300DATA'
#define CODE_EOF       '301EOF'
#define CODE_PUT        '302PUT'
#define CODE_REQUEST_FILENAME '303REQ_FILENAME'
```

```
#define CODE_REQUEST_ENCRYPT  '304REQ_ENCRYPT'
#define CODE_ENCRYPT     '305ENCRYPT'
#define CODE_REQUEST_FILE     '306REQUEST_FILE'

#define CODE_CMD       '400CMD'
#define CODE_LS        '401LS'

#define CODE_ERROR            '500ERROR'
#define CODE_ERROR_LS     '501ERROR_LS'
#define CODE_ERROR_CREAT_FILE '502ERROR_CREAT_FILE'

#define CODE_EXIT       '600EXIT'

#define MSG_ERR_WRONG_PROTOCOL  'Wrong protocol!'
#define MSG_ERROR_CREAT_FILE   'Couldn't create file'
#define MSG_PORT_NUMBER_ONLY   'Need port number only! \n%s <Port Number>'
#define MSG_ERR_COULDNT_SOCKET  'Couldn't obtain socket - %d'
#define MSG_ERR_COULDNT_CONNECT 'Couldn't connect!'
#define MSG_ERR_SENDING_DATA   'Couldn't send data!'
#define MSG_ERR_RECEIVING_DATA 'Couln't recieve data!'
#define MSG_ERR_CONNECTION_FAIL 'Connection failed.'
#define MSG_ERR_NO_DIR_STREAM  'Could not obtain the directory stream'

void debug(int debugLevel, char *fmt, ...);
void errorDoExit(char *fmt, ...);

#endif
```

Default.c:

```
/**
 * Author: Alejandro G. Carlstein
 * Description: File Transfer Client/Server
 */

#include 'default.h'

void debug(int debugLevel, char *fmt, ...){
 if (debugLevel == 1){
  va_list argp;
  fprintf(stdout, '[DBG] ');
  va_start(argp, fmt);
  vfprintf(stdout, fmt, argp);
  va_end(argp);
  fprintf(stdout, '\n');
 }
}

void errorDoExit(char *fmt, ...){
 va_list argp;
 fprintf(stderr, '[Error] ');
 va_start(argp, fmt);
 vfprintf(stderr, fmt, argp);
 va_end(argp);
 if (errno){
  fprintf(stderr, '=> %s\n', strerror(errno));
 }else{
  fprintf(stderr, '\n');
 }
 exit(1);
}
```

The following is the code for the server.
serv.c:

```
/**
 * Assignment: 1
 * Course: CS458
 * Author: Alejandro G. Carlstein
 * Description: FTP Server
 */

#include 'default.h'

/*max. length queue of pending connections may grow. */
#define MAX_BACKLOG       5
#define SETSOCKOPT_VAL      1

#define MSG_SERVER_NAME     'Server Name: ACARLSTEIN Server Version 1.0'
#define MSG_WAIT_CLIENT     'Waiting client...\n'
#define MSG_WAIT_CLIENT_ON_PORT '\nTCPServer Waiting for client on port %d\n'

#define MSG_ERR_NO_SOCKET_OPT  'Couldn't set and/or get socket options'
```

```
#define MSG_ERR_UNABLE_BIND     'Unable to bind'
#define MSG_ERR_UNABLE_LISTEN   'Unable to Listen'
#define MSG_ERR_CANT_SEND_LIST  'Can't send server list\n'

struct Connection{
 int sock;
 int bytes_recieved;
 int port_number;
 int socket_descriptor;
 char send_data[DATA_PACKET_SIZE];
 char recv_data[DATA_PACKET_SIZE];
  struct sockaddr_in server_addr;
  struct sockaddr_in client_addr;
};

void setUpConnection(struct Connection *new_connection,
           int port_number);
int menuDriver(struct Connection *new_connection);
void sendData(struct Connection *new_connection,
       const char* data);
int recieveData(struct Connection *new_connection);
void handShake(struct Connection* new_connection);
void sendListDirectoryContents(struct Connection* new_connection);
int receiveFileFromClient(struct Connection* new_connection);
int strdecrypt(const char* str_in,
        char* str_out);

/**
 * MAIN
 */
int main(int argc, char *argv[]){
 debug(DBG_LV0, 'argc: %d', argc);

 short DO_QUIT_PROGRAM;
 int i;
 int port_number;
 socklen_t sin_size = sizeof(struct sockaddr_in);
 struct Connection connection;

 for(i = 0; DBG_LV1 && i < argc; ++i)
  debug(DBG_LV1, 'argv[%d]: %s', i, argv[i]);

 if (argc > 2) errorDoExit(MSG_PORT_NUMBER_ONLY, argv[0]);

 port_number = (argc == 2) ? atoi(argv[1]) : DEFAULT_PORT;

 DO_QUIT_PROGRAM = QUIT_FALSE;
 while(DO_QUIT_PROGRAM){

  fflush(stdout);

  setUpConnection(&connection, port_number);
  printf(MSG_WAIT_CLIENT_ON_PORT, port_number);

  connection.socket_descriptor = accept(connection.sock,
                      (struct sockaddr *)&connection.client_addr,
                     &sin_size);

  if (connection.socket_descriptor  == -1)
   errorDoExit(MSG_ERR_COULDNT_CONNECT);

  handShake(&connection);

   menuDriver(&connection);

  close(connection.sock);

 }

  return 0;
}

/**
 * Menu driver waiting for instructions
 */
int menuDriver(struct Connection *new_connection){

 short DO_QUIT_CONNECTION;
 int bytes_received = 1;

 DO_QUIT_CONNECTION = QUIT_FALSE;
  while (DO_QUIT_CONNECTION){

  if(bytes_received == 0){
   DO_QUIT_CONNECTION = QUIT_TRUE;
  }else{

   printf(MSG_WAIT_CLIENT);

   bytes_received = recieveData(new_connection);
   if (strcmp(new_connection->recv_data, CODE_LS) == 0){
    sendListDirectoryContents(new_connection);
   }else
   if (strcmp(new_connection->recv_data, CODE_PUT) == 0){
    bytes_received = receiveFileFromClient(new_connection);
   }else{
    close(new_connection->sock);
    fprintf(stderr, MSG_ERR_WRONG_PROTOCOL ' - PUT\n');
    DO_QUIT_CONNECTION = QUIT_TRUE;
   }
  }
 }
 return 0;
}
```

```
/**
 * Set up connection with the client
 */
void setUpConnection(struct Connection *new_connection,
           int port_number){
 debug(DBG_LV0, 'setUpConnection(port_number: %d)', port_number);

 int opt_val = SETSOCKOPT_VAL;
 new_connection->port_number = port_number;

  if ((new_connection->sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  errorDoExit(MSG_ERR_COULDNT_SOCKET, new_connection->sock);

 if (setsockopt(new_connection->sock, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(int))
== -1)
  errorDoExit(MSG_ERR_NO_SOCKET_OPT);

 debug(DBG_LV1, 'Got socket!');

  new_connection->server_addr.sin_family = AF_INET;
  debug(DBG_LV1,'sin_family');

  new_connection->server_addr.sin_port = htons(port_number);
 debug(DBG_LV1,'sin_port');

  new_connection->server_addr.sin_addr.s_addr = INADDR_ANY;
 debug(DBG_LV1, 'sin_addr');

  bzero(&(new_connection->server_addr.sin_zero), NUM_BYTES);
 debug(DBG_LV1,'sin_zero');

 // bind a name to a socket
  if (bind(new_connection->sock,
      (struct sockaddr *)&new_connection->server_addr,
      sizeof(struct sockaddr)) == -1){
  close(new_connection->sock);
  errorDoExit(MSG_ERR_UNABLE_BIND);
 }

 // Listen for connection on the socket
  if (listen(new_connection->sock, MAX_BACKLOG) == -1){
  close(new_connection->sock);
    errorDoExit(MSG_ERR_UNABLE_LISTEN);
  }
 debug(DBG_LV1, 'Listening...');

}

/**
 * Send data to client
 */
void sendData(struct Connection *new_connection,
       const char* data){
```

```
 debug(DBG_LV0, 'sendData(data: %s)', data);

  int data_length;

 bzero(new_connection->send_data, DATA_PACKET_SIZE);
  strcpy(new_connection->send_data, data);
  data_length = strlen(new_connection->send_data);

  debug(DBG_LV1, 'new_connection->send_data: %s',
        new_connection->send_data);

 if (send(new_connection->socket_descriptor,
      new_connection->send_data,
      data_length, 0) != data_length){
  close(new_connection->sock);
  fprintf(stderr, MSG_ERR_SENDING_DATA '\n');
 }
}

/**
 * Receive data from client
 */
int recieveData(struct Connection *new_connection){
 debug(DBG_LV0, 'recieveData()');

 bzero(new_connection->recv_data, DATA_PACKET_SIZE);
 int bytes_recieved = recv(new_connection->socket_descriptor,
             new_connection->recv_data,
             DATA_PACKET_SIZE,
             0);

 debug(DBG_LV1, 'Check bytes received');

 if(bytes_recieved < 1){

  close(new_connection->sock);

  if(bytes_recieved == 0){
   fprintf(stderr, MSG_ERR_CONNECTION_FAIL '\n');
  }else{
   fprintf(stderr,MSG_ERR_RECEIVING_DATA '\n');
  }

 }else{

  debug(DBG_LV1, 'Received: %s', new_connection->recv_data);

  new_connection->bytes_recieved = bytes_recieved;
   new_connection->recv_data[bytes_recieved] = '\0';
  }

  return bytes_recieved;
}
```

```
/**
 * Hand shake with client to test own protocol
 */
void handShake(struct Connection* new_connection){
 debug(DBG_LV0, 'handShake()');

 printf('Connection from (%s , %d)...\n',
         inet_ntoa(new_connection->client_addr.sin_addr),
         ntohs(new_connection->client_addr.sin_port));

 debug(DBG_LV1,' If client say HELLO, Greed client with WELCOME');
  recieveData(new_connection);
 if (strcmp(new_connection->recv_data, CODE_HELLO) == 0){
  debug(DBG_LV1, 'Client handshake with server');
  sendData(new_connection, CODE_WELCOME);
 }else{
  close(new_connection->sock);
  fprintf(stderr, MSG_ERR_WRONG_PROTOCOL ' - HELLO/WELCOME \n');
 }

 debug(DBG_LV0,'If client ask for server name send server name to client');
  recieveData(new_connection);
 if (strcmp(new_connection->recv_data, CODE_REQ_SERVER_NAME) == 0){
  debug(DBG_LV1, 'Client asking for server name');
  sendData(new_connection, MSG_SERVER_NAME);
 }else{
  close(new_connection->sock);
  fprintf(stderr, MSG_ERR_WRONG_PROTOCOL ' - SERVER NAME\n');
 }
}

/**
 * Send list of directory contents to client
 */
void sendListDirectoryContents(struct Connection* new_connection){
 debug(DBG_LV0, 'void displayLocalListDirectoryContents()');
 int bytes_received;
 struct dirent *dirent_struct_ptr;
 DIR *directory_stream_ptr;

 if ((directory_stream_ptr = opendir('./')) != NULL){

   while ((dirent_struct_ptr = readdir(directory_stream_ptr))){
   sendData(new_connection, dirent_struct_ptr->d_name);

   bytes_received = recieveData(new_connection);
   if (strcmp(new_connection->recv_data, CODE_OK) != 0){
    fprintf(stderr, MSG_ERR_CANT_SEND_LIST);
   }
    }
  sendData(new_connection, CODE_EOF);
 }else{
  sendData(new_connection, CODE_ERROR_LS);
  errorDoExit(MSG_ERR_NO_DIR_STREAM);
 }

}

/**
 * Receive encrypted file from client
 */
int receiveFileFromClient(struct Connection* new_connection){
 debug(DBG_LV0, 'void receiveFileFromClient(is_encrypted: %s)');

 int bytes_received;

 char filename[DATA_PACKET_SIZE];
 char filename_se[DATA_PACKET_SIZE];
 char filename_sd[DATA_PACKET_SIZE];
 char str_unencrypted[DATA_PACKET_SIZE];
 FILE *fp_se, *fp_sd;

 /* Request filename */
 debug(DBG_LV1, 'Request filename from client');
 sendData(new_connection, CODE_REQUEST_FILENAME);

 /* receive filename */
 bytes_received = recieveData(new_connection);
 strcpy(filename, new_connection->recv_data);
 sprintf(filename_se, '%s_se', filename);
 sprintf(filename_sd, '%s_sd', filename);

 debug(DBG_LV1, 'filename: %s', filename);
 debug(DBG_LV1, 'filename_se: %s', filename_se);
 debug(DBG_LV1, 'filename_sd: %s', filename_sd);

 fp_se = fopen(filename_se, 'w');
 fp_sd = fopen(filename_sd, 'w');
 debug(DBG_LV1, 'Files open');

 if (fp_se == NULL || fp_sd == NULL){
  debug(DBG_LV1, 'Error, closing files');
  fclose(fp_se);
  fclose(fp_sd);
  errorDoExit(MSG_ERROR_CREAT_FILE);
 }else{

  debug(DBG_LV1, 'Download file...');
  /* Send code request file */
  sendData(new_connection, CODE_REQUEST_FILE);

  bytes_received = recieveData(new_connection);

   /* while loop until eof */
```

```
    while (strcmp(new_connection->recv_data, CODE_EOF) != 0){

    debug(DBG_LV1, 'Saving: %s', new_connection->recv_data);
     /* save encrypted package */
    fprintf(fp_se, '%s', new_connection->recv_data);

    /* Decencrypt package */
    bzero(str_unencrypted, DATA_PACKET_SIZE);
    strdecrypt(new_connection->recv_data, str_unencrypted);
    debug(DBG_LV1, 'Decrypt: %s\n', new_connection->recv_data);

    /* Save unecrypted package */
    debug(DBG_LV1, 'DECRYPT: %s', str_unencrypted);

    fprintf(fp_sd, '%s', str_unencrypted);

    debug(DBG_LV1, 'Sending OK...');
     /*send ok */
    sendData(new_connection, CODE_OK);
    bytes_received = recieveData(new_connection);
    }

  }

  fclose(fp_se);
  fclose(fp_sd);
  return 0;
}

/**
 * Decrypt string
 */
int strdecrypt(const char* str_in,
        char* str_out){
 debug(DBG_LV0, 'strdencrypt(str_in: %s)', str_in);

 int i, j;
 int i_cipher, i_key, i_temp;
 char str_key[DATA_PACKET_SIZE];
 strcpy(str_key, ENCRYPTION_KEY);
 char char_temp;
 for (i = 0, j = 0; i < strlen(str_in); ++i, ++j){

  i_cipher = str_in[i] - CHAR_CIPHER_A;
  i_key = str_key[j] - CHAR_KEY_A;
  i_temp = i_cipher - i_key;

  debug(DBG_LV1, 'Ci([%c]%d): %d, Ki([%c]%d): %d, Ci - Ki: %d',
     str_in[i], i, i_cipher, str_key[j], j, i_key, i_temp);

  if (i_temp > -1){
   //COMMON KNOWN DECRYPT ALGORIGHTM
   i_temp = ((i_temp) % 26);
  }else{
   //DECRYPT ALGORITHM FOR UNCOMMON CASES WHERE I_TEMP IS NEGATIVE
   i_temp = ((i_temp + 26) % 26);
  }

  char_temp = i_temp + CHAR_PLAIN_A;
  if (isalpha(char_temp)){
   str_out[i] = char_temp;
  }else{
   str_out[i] = '\0';
  }

  if ( (j + 1) >= strlen(str_key)) j = -1;
 }

 return 0;
}
```

The following is the code for the client.

cli.c

```
/**
 * Assignment: 1
 * Course: CS458
 * Author: Alejandro G. Carlstein
 * Description: Transfer Encrypted File Client
 */

#include 'default.h'

#define DEFAULT_HOST          'localhost'
#define EXIT_PROGRAM          0
#define DONT_EXIT_PROGRAM       1

#define CMD_QUIT           'quit'
#define CMD_HELP           'help'
#define CMD_LS            'ls'
#define CMD_LLS            'lls'
#define CMD_LPWD           'lpwd'
#define CMD_PUT            'put'

#define MSG_DISPLAY_MENU        'User help:\n'\
                   'help - Display help option\n'\
                    'lls  - Display local directory list contents\n'\
                                  'lpwd - Display local current directory\n'\
                                  'put  - Transfer local file to server\n'\
                                  '  put <file to transfer>\n'\
                                  'quit - Quit program\n'

#define MSG_ERROR_UNKNOWN_COMMAND   '[X] Command not recognized: %s\n'
#define MSG_ERROR_GETTING_HOSTNAME   'Couln't get hostname!'
#define MSG_ERROR_SERVER_DISCONNECTED 'Server disconnected...\n'
```

```
enum TOKENS{
 TOKEN_COMMAND,
 TOKEN_FILENAME,
 MAX_TOKENS
};

struct Connection{
 int sock;
 int bytes_received;
 int port_number;
 char send_data[DATA_PACKET_SIZE];
 char recv_data[DATA_PACKET_SIZE];
 struct hostent *host;
  struct sockaddr_in server_addr;
};

void getConnection(struct Connection *new_connection,
          char *hostname,
          int port_number);
void menuDriver(struct Connection *new_connection);
int receiveData(struct Connection *new_connection);
int sendData(struct Connection* new_connection, const char* data);
void handShake(struct Connection* new_connection);
void promptUser(char* tokens[]);
void displayHelp(void);
void displayLocalListDirectoryContents(void);
void printNameCurrentDirectory(void);
void receiveListDirectoryContents(struct Connection *new_connection);
void putFileOnServer(struct Connection *new_connection,
           const char* filename);
int strencrypt(const char* str_in,
               char* str_out);
/**
 * MAIN
 */
int main(int argc, char* argv[]){
 debug(DBG_LV0, 'argc: %d', argc);

 int i;
 int port_number = (argc >= 3 ) ? atoi(argv[2]) : DEFAULT_PORT;
 char* host_name = (argc >= 2) ? argv[1] : DEFAULT_HOST;

 struct Connection connection;

 for(i = 0; i < argc; ++i)
  debug(DBG_LV0, 'argv[%d]: %s', i, argv[i]);

 if (argc > 3)
  errorDoExit(MSG_PORT_NUMBER_ONLY, argv[0]);

 debug(DBG_LV1, 'host: %s', host_name);
 getConnection(&connection, host_name, port_number);

 handShake(&connection);

 menuDriver(&connection);

 return 0;

}

/**
 * Driver menu
 */
void menuDriver(struct Connection *new_connection){
 debug(DBG_LV0, 'menuDriver()');

 short doExit = DONT_EXIT_PROGRAM;
 int i;
 char* tokens[MAX_TOKENS] = {NULL, NULL};
 char command[DATA_PACKET_SIZE];
 char filename[DATA_PACKET_SIZE];

  while(doExit){

  promptUser(tokens);

  for (i = 0; DBG_LV1 && i < MAX_TOKENS; ++i)
   debug(DBG_LV1, 'TOKEN[%d]: %s', i, tokens[i]);

  /* Must copy string to work on SunOS */
  if (tokens[TOKEN_COMMAND] != NULL){
   strcpy(command, tokens[TOKEN_COMMAND]);
     debug(DBG_LV1, '[Command: %s]', command);
  }

  if (tokens[TOKEN_FILENAME] != NULL){
     strcpy(filename, tokens[TOKEN_FILENAME]);
     debug(DBG_LV1, '[Filename: %s]', filename);
    }

  /* MENU */

  if (strcmp(command, CMD_QUIT) == 0){
   debug(DBG_LV1, 'COMMAND> %s: ', CMD_QUIT);
   close(new_connection->sock);
   printf('Bye Bye\n');
   doExit = EXIT_PROGRAM;

  }else
  if (strcmp(command, 'help') == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_HELP);
   displayHelp();
  }else
  if (strcmp(command, CMD_LLS) == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_LLS);
   displayLocalListDirectoryContents();

  }else
  if (strcmp(command, CMD_LLS) == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_LLS);
   displayLocalListDirectoryContents();

  }else
  if (strcmp(command, CMD_LS) == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_LS);
   receiveListDirectoryContents(new_connection);

  }else
  if (strcmp(command, CMD_LPWD) == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_LPWD);
   printNameCurrentDirectory();

  }else
  if (strcmp(command, CMD_PUT) == 0){
   debug(DBG_LV1, 'COMMAND> %s ', CMD_PUT);
   putFileOnServer(new_connection, filename);

  }else{
   printf(MSG_ERROR_UNKNOWN_COMMAND, command);
  }

 }

}

/**
 * Obtain connection with server
 */
void getConnection(struct Connection *new_connection,
          char *hostname,
          int port_number){
 debug(DBG_LV0, 'getConnection(hostname: %s, port_number: %d)', hostname, port_number);

 if ((new_connection->host = gethostbyname(hostname)) == NULL)
  errorDoExit(MSG_ERROR_GETTING_HOSTNAME);

 new_connection->port_number = port_number;

 if ((new_connection->sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  errorDoExit(MSG_ERR_COULDNT_SOCKET, new_connection->sock);

 debug(DBG_LV1, 'Got socket!');

  new_connection->server_addr.sin_family = AF_INET;
 debug(DBG_LV1,'sin_family');
```

```
  new_connection->server_addr.sin_port = htons(port_number);
 debug(DBG_LV1,'sin_port');

  new_connection->server_addr.sin_addr = *((struct in_addr *)new_connection->host-
>h_addr);
 debug(DBG_LV1, 'sin_addr');

 // bzero() is used only for setting the values to zero
 bzero(&(new_connection->server_addr.sin_zero), NUM_BYTES);
 debug(DBG_LV1,'sin_zero');

  if (connect(new_connection->sock,
         (struct sockaddr *)&new_connection->server_addr,
         sizeof(struct sockaddr)) == -1)
  errorDoExit(MSG_ERR_COULDNT_CONNECT);

 debug(DBG_LV1, 'Got connection!');
}

/**
 * Receive data
 * @Return: Number of bytes received
 */
int receiveData(struct Connection *new_connection){
 debug(DBG_LV0, 'receiveData()');

 bzero(new_connection->recv_data, DATA_PACKET_SIZE);
 int bytes_received = recv(new_connection->sock,
              new_connection->recv_data,
              DATA_PACKET_SIZE,
              0);

 if(bytes_received < 1){
  close(new_connection->sock);
  if(bytes_received == 0){
   close(new_connection->sock);
   printf(MSG_ERROR_SERVER_DISCONNECTED);
   errorDoExit(MSG_ERR_CONNECTION_FAIL);

  }else{
   errorDoExit(MSG_ERR_RECEIVING_DATA);
  }
 }

 debug(DBG_LV1, 'Received (%d): %s',
    strlen(new_connection->recv_data), new_connection->recv_data);

 new_connection->bytes_received = bytes_received;
  new_connection->recv_data[bytes_received] = '\0';

  return bytes_received;
}
```

```
/**
 * Send data
 * @Return: Number of bytes sended
 */
int sendData(struct Connection* new_connection,
      const char* data){
 debug(DBG_LV0, 'sendData(%s)', data);
 int data_length;

 bzero(new_connection->send_data, DATA_PACKET_SIZE);
 strcpy(new_connection->send_data, data);
 data_length = strlen(new_connection->send_data);

 if (send(new_connection->sock, new_connection->send_data,
      data_length, 0) != data_length){
  close(new_connection->sock);
  errorDoExit(MSG_ERR_SENDING_DATA);
 }

 return data_length;
}

/**
 * Hand shake with the server to verify own protocol
 */
void handShake(struct Connection* new_connection){
 debug(DBG_LV0, 'handShake()');

 debug(DBG_LV1, 'Sending HELLO message...');
 sendData(new_connection, CODE_HELLO);

 debug(DBG_LV1, 'Getting WELCOME message...');
 receiveData(new_connection);

 if (strcmp(new_connection->recv_data, CODE_WELCOME) > -1){
  debug(DBG_LV1, 'Got WELCOME!');

  debug(DBG_LV1, 'Request server name');
  sendData(new_connection, CODE_REQ_SERVER_NAME);

  receiveData(new_connection);
  printf('Server Name: %s\n', new_connection->recv_data);
 }else{
  errorDoExit(MSG_ERR_WRONG_PROTOCOL);
 }
}

/**
 * Prompt user for input
 */
void promptUser(char* tokens[]){
 char str_input[DATA_PACKET_SIZE];
 char *p;
 int i;

 printf('\nftp> ');
 fgets(str_input, DATA_PACKET_SIZE, stdin);

 p = strtok(str_input, ' \n');
  for (i = 0; p != NULL; (p = strtok(NULL, ' \n')), i++)
  tokens[i] = p;
}

/**
 * Display help message
 */
void displayHelp(void){
 debug(DBG_LV0, 'displayHelp()');
 printf(MSG_DISPLAY_MENU);
}

/**
 * Display local list directory contents
 */
void displayLocalListDirectoryContents(void){
 debug(DBG_LV0, 'void displayLocalListDirectoryContents()');

 struct dirent *dirent_struct_ptr;
 DIR *directory_stream_ptr;

 if ((directory_stream_ptr = opendir('./')) != NULL){
  printNameCurrentDirectory();

   while ((dirent_struct_ptr = readdir(directory_stream_ptr))){
    printf(' %s\n',dirent_struct_ptr->d_name);
    }
 }else{
  errorDoExit(MSG_ERR_NO_DIR_STREAM);
 }

}

/**
 * Print the name of the current directory
 */
void printNameCurrentDirectory(void){
 debug(DBG_LV0, 'void printNameCurrentDirectory()');

 long size;
 char *buf;
 char *ptr;
 size = pathconf('.', _PC_PATH_MAX);
 if ((buf = (char *)malloc((size_t)size)) != NULL){
  ptr = getcwd(buf, (size_t)size);
 }else{
  errorDoExit('Could not obtain the current directory');
 }
 printf('Current Directory: %s\n', ptr);
}

/**
 * Receive the list of directory contents from the server
 */
void receiveListDirectoryContents(struct Connection *new_connection){
 debug(DBG_LV0, 'receiveListDirectoryContents()');

 sendData(new_connection, CODE_LS);

 receiveData(new_connection);
 while (strcmp(new_connection->recv_data, CODE_EOF) != 0){

  if (strcmp(new_connection->recv_data, CODE_ERROR_LS) == 0){
   fprintf(stderr, 'Couldn't read list directory contents!\n');
   break;
  }else{
   printf('%s\n', new_connection->recv_data);
   sendData(new_connection, CODE_OK);
  }
  receiveData(new_connection);
 }

}

/**
 * Put an encrypted file on the server
 */
void putFileOnServer(struct Connection *new_connection,
           const char* filename){
 debug(DBG_LV0, 'void putFileOnServer(filename: %s)', filename);

 char str_filename[DATA_PACKET_SIZE];
 char filename_ce[DATA_PACKET_SIZE];
 char str_in_file[DATA_PACKET_SIZE];
 char str_in_file_encrypt[DATA_PACKET_SIZE];
 FILE *fp, *fp_ce;

 strcpy(str_filename, filename);

 debug(DBG_LV1, 'str_filename: %s', str_filename);

 sprintf(filename_ce, '%s_ce', str_filename);

 debug(DBG_LV1, 'filename_ce: %s', filename_ce);

 debug(DBG_LV1, 'OPEN FILE TO READ');
 fp = fopen(str_filename, 'r');

 debug(DBG_LV1, 'OPEN FILE TO WRITE');
```

```
fp_ce = fopen(filename_ce, 'w');

if (fp == NULL || fp_ce == NULL){
 fclose(fp_ce);
 fclose(fp);
 errorDoExit('Couldn't open file');
}else{

 /* Send PUT code to server */
 sendData(new_connection, CODE_PUT);

 debug(DBG_LV0, 'Waiting for CODE_REQUEST_FILENAME');
 /* Receive filename request form server */
 receiveData(new_connection);
 if (strcmp(new_connection->recv_data, CODE_REQUEST_FILENAME) == 0){

  debug(DBG_LV0, 'Send filename');
  /* Send filename to server */
  sendData(new_connection, str_filename);

  /* Receive request to send file */
  receiveData(new_connection);
  if (strcmp(new_connection->recv_data, CODE_REQUEST_FILE) == 0){
   printf('Sending file: %s.\n', filename);
    /* Encrypt file */
   while(fgets(str_in_file, DATA_PACKET_SIZE, fp) != NULL){
    debug(DBG_LV2, 'Reading (Len: %d): %s', strlen(str_in_file), str_in_file);

    /* encrypt package */
    strencrypt(str_in_file, str_in_file_encrypt);

    /* Save it in <filename>_ce */
    fprintf(fp_ce, '%s', str_in_file_encrypt);

    /* send packages */
    sendData(new_connection, str_in_file_encrypt);

    /* Waiting for ok */
    receiveData(new_connection);
    if (strcmp(new_connection->recv_data, CODE_OK) == 0){
     printf('OK');
    }else{
     errorDoExit('Wrong protocol - Waiting OK');
    }

   }
   sendData(new_connection, CODE_EOF);

  }else{
   errorDoExit('Wrong protocol - Waiting request for file ');
  }

 }else{
    errorDoExit('Wrong protocol - Waiting for request of filename');
   }

  }

  fclose(fp);
  fclose(fp_ce);

}

/**
 * Encrypt the string using key
 * @Return: Number of bytes encrypted
 */
int strencrypt(const char* str_in,
               char* str_out){
 debug(DBG_LV0, 'strencrypt(str_in: %s)', str_in);

 int i, j;
 int i_plain, i_key;

 char str_key[DATA_PACKET_SIZE];

 strcpy(str_key, ENCRYPTION_KEY);

 for (i = 0, j = 0; i < strlen(str_in) - 1; ++i, ++j){

  i_plain = (unsigned int)str_in[i] - CHAR_PLAIN_A;
  i_key = (unsigned int)str_key[j] - CHAR_KEY_A;

  str_out[i] = (char)((i_plain + i_key) % 26 + CHAR_CIPHER_A);

  debug(DBG_LV1, '(str_out: %d) [%c:%d] = (str_in:%d)[%c:%d], (key:%d)[%c:%d]',
     i, str_out[i], (int)str_out[i],
     i, str_in[i], (int) str_in[i],
     j, str_key[j], (int)str_key[j]);

  if ( (j + 1) >= strlen(str_key)) j = -1;
 }

 return strlen(str_in);
}
```

When I wrote the code, I was following certain guidelines. As you may think the code could be written better. I agree.

In case you have any questions about the code please post them. I will try to answer you as soon as possible.