Contents

1 Basic Test Results

```
g++ -Wall -Wvla -Wextra -g -std=c++11 -c helper.cpp
g++ -Wall -Wvla -Wextra -g -std=c++11 -lpthread -o srftp srftp.cpp helper.o
g++ -Wall -Wvla -Wextra -g -std=c++11 -o clftp clftp.cpp helper.o
rm -f sftrp clftp *.o
g++ -Wall -Wvla -Wextra -g -std=c++11 -c helper.cpp
g++ -Wall -Wvla -Wextra -g -std=c++11 -o clftp clftp.cpp helper.o
g++ -Wall -Wvla -Wextra -g -std=c++11 -lpthread -o srftp srftp.cpp helper.o
rm -f sftrp clftp *.o
### Looking for Missing Files: ###
### README testing ###
Reading /tmp/bodek.PddBJI/os/EX5_Submission/roeia1/presubmission/testdir/testTmp/README
### Makefile testing ###
### End of Presubmission Testing ###
```

2 README

```
roeia1, maayanimi
Roei Avrahami (304986714), Maayan Mizrahi (304901259)
EX: 5

FILES:

clftp.cpp            - The client.
srftp.cpp            - The server.
helper.h            - Helper file which contain functions and consts.
helper.cpp            - Implementation of helper.h
Makefile            - The makeFile.
README              - this file.
performance.jpg        - Performance evaluation.

REMARKS:

In this ex we're implementing a TCP protocol for sending files from
a client to a server.
First we sent the file size for validating if it is a legal size for the server.
Then, we sent the new name of the file that will be created in the server
(first the name size and then the name itself),
and after we send the content of the file.
We sent the data in packets of 4096 bytes.
After we finished sending the file we closed the client socket.

ANSWERS:

Implementing using UDP connection :
    Our UDP protocol will use message numbering identification
    (IP and port num as we learned in the tirgul, and the packet number).
    This identification will let the receiver know from where the message from
    and the number of the packet.
    The sender will know the message arrived.
    After connection being established, the client will send the number
    of packets that he is going to send to the server.
    Then, the client will wait for a ack from the server,
    and if the ack wasn't received the client will send it again.
    Once the ack confirmed at the client, he will start sending the
    packets 5 each time when each packet will be numbered.
    After sending those 5 packets he wait for ack before sending the next 5,
    if no ack he will send all of them again.
    If the ack won't be ok it will say which packets were lost and the client
    will send them again plus the next packets that the sum will be always 5
    (for example if 3 lost he will send those 3 again with 2 squential packets
    of the previous 5).
    Finally, if the ack is ok it will next the next 5 packets.
    From the server side, for each 5 packets he receives he will check if
    the packet's numbers sequential to the previous last packet that already received,
    or one of the "lost" packets he needs to fill.
    If still there are still "lost" packets needed to be fill, the server
    will send to the client the numbers of these packets with the ack,
    if not he will send an ok ack.
    This way if there are 5 "lost" packets he will request all of them
    and the client will only send these 5 again and wont send new ones.
    If a defined time passed and the server don't get packets
    from the client, it will re send the previous ack to the client again.
    When the server received all the packets according to the number
    that received in the beginning, the socket will be closed.
Efficiency:
```

```
60      In this way we are sending 5 packets at a time, and if some
61      packets "lost" in the UDP we will send those the next time,
62      while completing the 5 packets with new ones if those that "lost"
63      are less then 5.
64      This is more efficient then waiting for ack for each packet
65      because we sending more data each time
66      (assuming that there will be a few "lost" packets).
67  Differences:
68      In this ex we send the file's size, but in the UDP protocol
69      we send the num of packets that are going to be sent by the client
70      and verify in the end that all these packets arrived.
71      Another difference is the waiting and ack every 5 packets
72      and recovering "lost" packets by requesting those again.
73
74  Analysis:
75      In the graph we can clearly see that the dependency of the transmission
76      time in the file size is linear, meaning that the bigger the file gets,
77      the transmission time will be longer in a linear way.
78      In conclusion, the growth rate of the transmission time will be equal
79      for different file sizes since its linear.
```

3 Makefile

```
CXX=g++
FLAGS= -Wall -Wvla -Wextra -g -std=c++11

all: srftp clftp

srftp : srftp.cpp helper.o
    ${CXX} ${FLAGS} -lpthread -o $@ $^

helper.o: helper.cpp helper.h
    ${CXX} ${FLAGS} -c $<

clftp: clftp.cpp helper.o
    ${CXX} ${FLAGS} -o $@ $^

clean:
    rm -f sftrp clftp *.o

tar: ex5.tar

ex5.tar: srftp.cpp clftp.cpp helper.h helper.cpp README Makefile
    tar -cvf $@ $^

.PHONY: tar, clean, all
```

4 clftp.cpp

```cpp
/*
 * clftp.cpp
 *
 *  Created on: Jun 9, 2015
 *      Author: roeia1
 */

#include "helper.h"

#define PARAM_NUM 5
#define SERVER_PORT 1
#define SERVER_HOST_NAME 2
#define FILE_TO_TRANSFER 3
#define FILENAME_IN_SERVER 4
#define CLIENT_ERROR_MSG "Usage: clftp server-port server-hostname file-to-transfer filename-in-server"
#define BIG_FILE_MSG "Transmission failed: too big file"

/*
 * Sending the file data
 */
void sendFileData(int sock, int fileSize, ifstream& ifs)
{
    char* buffer = (char*)malloc(PACKET_SIZE);
    if (buffer == NULL)
    {
        error("malloc");
        exit(EXIT_FAILURE);
    }
    int bytesToSend = fileSize;
    while (bytesToSend > PACKET_SIZE)
    {
        ifs.read(buffer, PACKET_SIZE);
        sendBuffer(sock, buffer, PACKET_SIZE, false);
        bytesToSend -= PACKET_SIZE;
    }
    if (bytesToSend != 0)
    {
        ifs.read(buffer, bytesToSend);
        sendBuffer(sock, buffer, bytesToSend, false);
    }
    free (buffer);
}

/*
 * Getting the file size
 */
int getFileSize(ifstream &ifs)
{
    long begin = ifs.tellg();
    ifs.seekg (0, ios::end);
    long end = ifs.tellg();
    ifs.seekg(ios::beg);
    return end - begin;
}

int main(int argc , char *argv[])
{
    int serverPort = stringToInt(argv[SERVER_PORT], CLIENT_ERROR_MSG);
    ifstream fileToTransfer(argv[FILE_TO_TRANSFER], ifstream::in);
```

```
60      // Checking num of args, if port legal and if the file exists
61      if (argc != PARAM_NUM || serverPort < MIN_PORT_NUM || serverPort > MAX_PORT_NUM ||
62          strlen(argv[FILENAME_IN_SERVER]) > NAME_MAX || fileToTransfer == NULL)
63      {
64          cout << CLIENT_ERROR_MSG << endl;
65          exit(EXIT_FAILURE);
66      }
67      int fileSize = getFileSize(fileToTransfer);
68      int sock;
69      // Init the address
70      struct sockaddr_in sa;
71      struct hostent* hp;
72      hp = gethostbyname(argv[SERVER_HOST_NAME]);
73      memset(&sa, 0, sizeof(sa));
74      sa.sin_family = hp->h_addrtype;
75      memcpy((char*)&sa.sin_addr, hp->h_addr, hp->h_length);
76      sa.sin_port = htons((u_short)serverPort);
77      // Create socket
78      if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
79      {
80          error("socket");
81          exit(EXIT_FAILURE);
82      }
83      //Connect to remote server
84      if (connect(sock , (struct sockaddr*)&sa , sizeof(sa)) < 0)
85      {
86          cerr << strerror(errno) << endl;
87          close(sock);
88          return 1;
89      }
90      sendBuffer(sock, (char*)&fileSize, sizeof(int), false);
91      char checkSize[1];
92      if (recv(sock, checkSize, 1, 0) == ERROR)
93      {
94          error("recv");
95          exit(EXIT_FAILURE);
96      }
97      if (checkSize[0] == SUCCESS)
98      {
99          // Sending the file name in server size
100         int nameSize = strlen(argv[FILENAME_IN_SERVER]);
101         sendBuffer(sock, (char*)&nameSize, sizeof(int), false);
102         // Sending the file name in server
103         sendBuffer(sock, argv[FILENAME_IN_SERVER], nameSize, false);
104         // Sending the file data
105         sendFileData(sock, fileSize, fileToTransfer);
106     }
107     else
108     {
109         cout << BIG_FILE_MSG << endl;
110     }
111     close(sock);
112     fileToTransfer.close();
113     return 0;
114 }
```

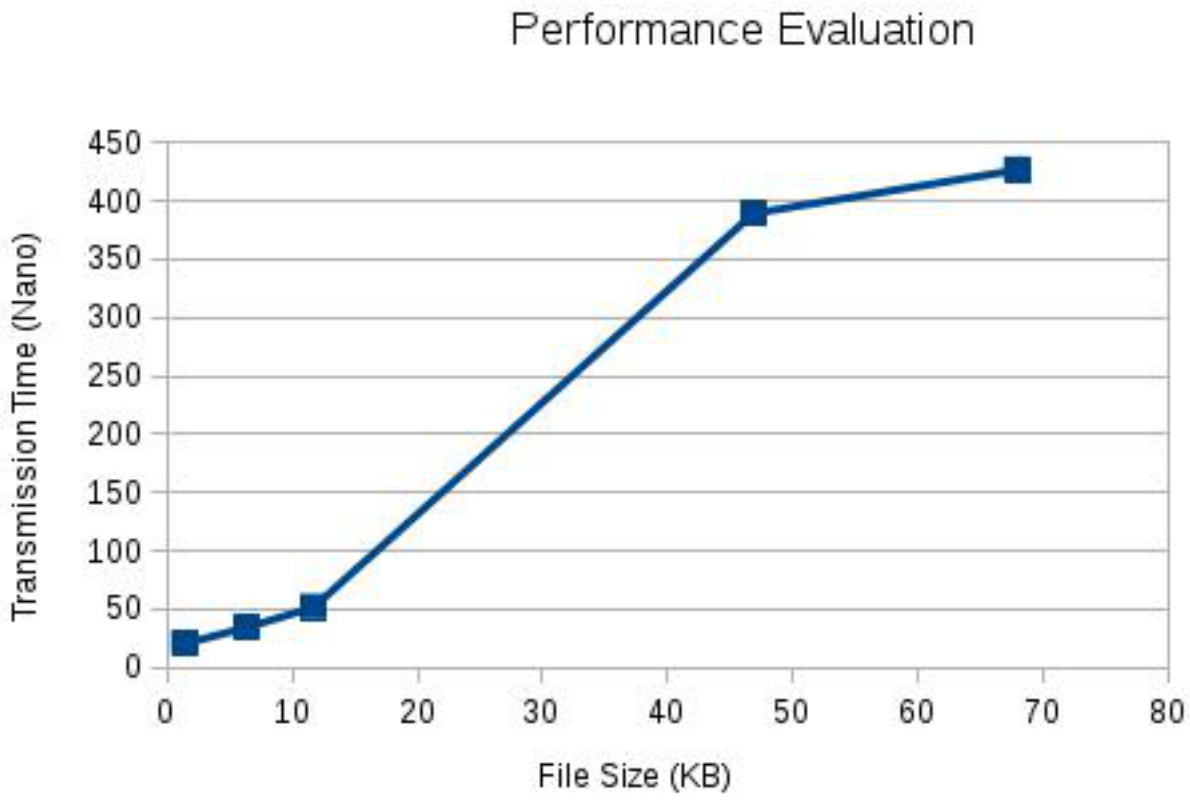
5 helper.h

```
1  /*
2   * helper.h
3   *
4   *  Created on: Jun 11, 2015
5   *      Author: roeia1
6   */
7
8  #ifndef HELPER_H_
9  #define HELPER_H_
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <sys/time.h>
14 #include <sys/socket.h>
15 #include <arpa/inet.h>
16 #include <pthread.h>
17 #include <iostream>
18 #include <sys/types.h>
19 #include <netinet/in.h>
20 #include <sys/param.h>
21 #include <string>
22 #include <netdb.h>
23 #include <unistd.h>
24 #include <fstream>
25 #include <cerrno>
26 #include <limits.h>
27 #include <stdexcept>
28 #include <errno.h>
29 #include <cstring>
30
31 using namespace std;
32
33 #define PACKET_SIZE 4096
34 #define MAX_PORT_NUM 65535
35 #define MIN_PORT_NUM 1
36 #define SUCCESS '1'
37 #define ERROR -1
38
39 /*
40  * This function represent a system call error,
41  * receiving the name of the system call printing to cerr.
42  */
43 void error(string systemCall);
44
45 /*
46  * Sending data through a buffer
47  */
48 void sendBuffer(int sock, char* buffer, int size,  bool isThread);
49
50 /*
51  * Converting string to int, if fails print
52  */
53 int stringToInt(char* input, string errorMsg);
54
55 #endif /* HELPER_H_ */
```

6 helper.cpp

```cpp
/*
 * helper.cpp
 *
 *  Created on: Jun 11, 2015
 *      Author: roeia1
 */

#include "helper.h"

/*
 * This function represent a system call error,
 * receiving the name of the system call printing to cerr.
 */
void error(string systemCall)
{
    cerr << "Error: function:" << systemCall << " errno:" << strerror(errno) << ".\n" << endl;
}

/*
 * Sending data through a buffer
 */
void sendBuffer(int sock, char* buffer, int size, bool isThread)
{
    int bytesSent = 0;
    int sent;
    while (bytesSent < size)
    {
        sent = send(sock, buffer + bytesSent, size - bytesSent, 0);
        if (sent == ERROR)
        {
            error("send");
            if (isThread)
            {
                pthread_exit(NULL);
            }
            exit(EXIT_FAILURE);
        }
        bytesSent += sent;
    }
}

/*
 * Converting string to int, if fails print
 */
int stringToInt(char* input, string errorMsg)
{
    int num;
    try
    {
        num = std::stoi(input);
    }
    catch (exception &e)
    {
        cout << errorMsg << endl;
        exit(EXIT_FAILURE);
    }
    return num;
}
```

7 performance.jpg



8 srftp.cpp

```
1  /*
2   * srftp.cpp
3   *
4   *  Created on: Jun 9, 2015
5   *      Author: roeia1
6   */
7  #include "helper.h"
8
9  #define PARAM_NUM 3
10 #define SERVER_PORT 1
11 #define MAX_FILE_SIZE 2
12 #define SERVER_ERROR_MSG "Usage: srftp server-port max-file-size"
13 #define MAX_LISTEN 5
14
15 int maxFileSize;
16
17 /*
18  * Receiving the file data and creating a new file with the same data.
19  */
20 void recvFileData(int sock, int fileSize, ofstream& fileToCreate)
21 {
22     char* buffer = (char*)malloc(PACKET_SIZE);
23     if (buffer == NULL)
24     {
25         error("malloc");
26         pthread_exit(NULL);
27     }
28     int bytesToWrite = fileSize;
29     while (bytesToWrite > PACKET_SIZE)
30     {
31         recv(sock, buffer, PACKET_SIZE, 0);
32         fileToCreate.write(buffer, PACKET_SIZE);
33         bytesToWrite -= PACKET_SIZE;
34     }
35     if (bytesToWrite != 0)
36     {
37         recv(sock, buffer, bytesToWrite, 0);
38         fileToCreate.write(buffer, bytesToWrite);
39     }
40     free (buffer);
41 }
42
43 /*
44  * This function will handle connection for each client to the server.
45  */
46 void* clientHandler(void* sockDesc)
47 {
48     // Get the socket descriptor
49     int sock = *(int*)sockDesc;
50     // Getting the file size from the client
51     char fileSize[sizeof(int)];
52     if (recv(sock, fileSize, sizeof(int), 0) == ERROR)
53     {
54         error("recv");
55         pthread_exit(NULL);
56     }
57     char res[1];
58     // Sending the client if the file size is ok
59     int nFileSize = *((int*)fileSize);
```

```
60      (nFileSize <= maxFileSize) ? (res[0] = '1') : (res[0] = '0');
61      sendBuffer(sock, res, 1, true);
62      // If the file size ok creating the file
63      if (res[0] == '1')
64      {
65          // Getting the file name size
66          char nameSize[sizeof(int)];
67          if (recv(sock, nameSize, sizeof(int), 0) == ERROR)
68          {
69              error("recv");
70              pthread_exit(NULL);
71          }
72          // Getting the file name
73          int nNameSize = *((int*)nameSize);
74          char* fileName = new char[nNameSize + 1];
75          if (recv(sock, fileName, nNameSize, 0) == ERROR)
76          {
77              delete fileName;
78              error("recv");
79              pthread_exit(NULL);
80          }
81          fileName[nNameSize] = '\0';
82          ofstream fileToCreate(fileName, ofstream::out);
83          recvFileData(sock, nFileSize, fileToCreate);
84          delete fileName;
85      }
86      return 0;
87  }
88
89  int main(int argc , char *argv[])
90  {
91      int serverPort = stringToInt(argv[SERVER_PORT], SERVER_ERROR_MSG);
92      // Checking num of args, if port legal and if legal max file size
93      if (argc != PARAM_NUM || serverPort < MIN_PORT_NUM || serverPort > MAX_PORT_NUM ||
94          ((maxFileSize = stringToInt(argv[MAX_FILE_SIZE], SERVER_ERROR_MSG)) < 0))
95      {
96          cout << SERVER_ERROR_MSG << endl;
97          exit(EXIT_FAILURE);
98      }
99      //Prepare the sockaddr_in structure
100     char myHostName[MAXHOSTNAMELEN+1];
101     struct sockaddr_in sa;
102     struct hostent* hp;
103     memset(&sa,0,sizeof(struct sockaddr_in));
104     gethostname(myHostName, MAXHOSTNAMELEN);
105     hp = gethostbyname(myHostName);
106     sa.sin_family = hp->h_addrtype;
107     memcpy(&sa.sin_addr, hp->h_addr, hp->h_length);
108     sa.sin_port = htons((u_short)serverPort);
109     //Create socket
110     int sockListen;
111     if ((sockListen = socket(AF_INET, SOCK_STREAM, 0)) < 0)
112     {
113         error("socket");
114         exit(EXIT_FAILURE);
115     }
116     //Bind
117     if(bind(sockListen,(struct sockaddr*)&sa, sizeof(struct sockaddr_in)) < 0)
118     {
119         close(sockListen);
120         error("bind");
121         exit(EXIT_FAILURE);
122     }
123     //Listen
124     listen(sockListen, MAX_LISTEN);
125     int clientSock;
126     while(true)
127     {
```

```
128         if((clientSock = accept(sockListen, NULL, NULL)) < 0)
129         {
130             error("accept");
131             exit(EXIT_FAILURE);
132         }
133         pthread_t clientThread;
134         int* newSock = (int*)malloc(sizeof(int));
135         *newSock = clientSock;
136         if(pthread_create(&clientThread, NULL, clientHandler, (void*)newSock) < 0)
137         {
138             error("pthread_create");
139             exit(EXIT_FAILURE);
140         }
141     }
142     return 0;
143 }
```