Contents

1 Basic Test Results 2

2 README 3

3 Makefile 5

4 clftp.cpp 6

5 iosafe.h 9

6 iosafe.cpp 11

7 performance.jpg 14

8 srftp.cpp 16

1 Basic Test Results

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

2 README

```
akiva_s, roigreenberg
Akiva Sygal(305277220), Roi Greenberg(305571234)
Ex: 5

FILES:
iosafe.h - header file for iosafe.cpp
iosafe.cpp - a helper with functions that read and write buffers
clftp.cpp -  as described in the excercise instructions
srftp.cpp -  as described in the excercise instructions
README
Makefile

REMARKS:
iosafe contains some code that is used to send information safely between client and server.
it implements some functions that read from the buffer the size, name, and file information itself,
based on the agreed format of buffer information between client and server.
the rest is implemented by multi-threading read and write through TCP sockets.

ANSWERS:
***UDP IMPLEMENTATION***
if we want to implement a UDP-based protocol, first we want our sockets not to establish a
safe connection using "listen" and "accept", but just after the "bind" operation to start
sending and getting pre-defined size datagrams (can set it to 512 bytes each one, for example)
of information using "send" and "recivefrom".
the datagrams of information can be lost while sending, or arrive in a different order.
to solve that, the client should re-format the information such that each of our packets would
start with the packet serial number (before the actual bits of informationit carries) in
its first 4 bytes, a fact known also by the reciever (server). the client stores the information
in the packets he gets in a pre-sized array, with enough space for a pre-defined number of packets
X (if we want this number to be efficient, we can decide it using the algorithm for window size
learned in class, the one that the TCP protocol uses with exponential-limited size),
*according to their order of serial numbers attached* (not to the order of recieveing them).
when this array is full, the server sends some "ack" saying the array is full and the client
should now pend the sending of new packets (if the number X is constant, the client can know
it in advance and every X packets stop and wait for an "ack"). after this ack, the client
knows it now waits to know if it can continue to send new packets. the server then scans its
array, looking for missing packets. then it sends the client the numbers of missing packets
and the client sends them again, and again, till it gets from the server the second ack saying
all packets have arrived and the arry is full. before server sends the second ack, it empties
the array, writing all its content to the file. after getting the second ack the client knows
it can start sending the next X packets, that will fill the empty array of the server,
and the process statrts again.
comment: in UDP we are not sure if a packet would arrive, or when will it arrive, but once we
got it we know it's probably correct because of the checksum.
another comment: if server gets a packet that he already have stored in the array, it just throws
it to garbage.
we see that in UDP-using protocol, the implementation of reliable server is much more complicated
and less efficient from our side (even though we say we can try to immitate the efficient algorithm
that TCP is using to determine its window size).we cannot trust that everything that was sent would
actually arrive, so we need to check every X packets that we've got all of them. second, we can't
just write to file each packet we get since the order is messy, and for that porpuse we need the
helper-array to store the packets in their actual order (for the same reason, we need the packets
to have serial numbers). in the TCP implementation, we did not need to worry about the dividing
into packets - it was an internal issue of the TCP protocol, we just sent a full buffer and read
it fully once, without trusting about the little units that the information is divided into while
sending.


***PERFORMANCES***
```

```
Size in bytes:    Time in nano-seconds:
896               1582835
43990               1955767
167225               3383060
430951               6781904
1506422               14690124

2938880               26281373
6516529               55507357
11422588      347430763
```

this gets somehow direct proportion between the file size and the time it is being sent.
those results was get by the average of 250 runs on each one of the files, calling "get time of the day" before and after sending, relying on the fact that in TCP the program would not continue before the whole information was recieved successfully.

3 Makefile

```
CC = g++ -Wall
FLAG = -std=c++11
LIBSRC = srftp.cpp clftp.cpp iosafe.cpp iosafe.h

all: srftp clftp




srftp.o: iosafe.h
	$(CC) $(FLAG) srftp.cpp -c

clftp.o: iosafe.h
	$(CC) $(FLAG) clftp.cpp -c


srftp: srftp.o
	$(CC) $(FLAG)  srftp.o iosafe.cpp -lpthread -L./ -o srftp

clftp: clftp.o
	$(CC) $(FLAG) clftp.o iosafe.cpp -o clftp

TAR=tar
TARFLAGS=-cvf
TARNAME=ex5.tar
TARSRCS=$(LIBSRC) Makefile README performance.jpg

tar:
	$(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)

clean:
	$(RM) *.o srftp clftp  *~ *core


.PHONY: clftp clftp.o srftp srftp.o iosafe.o
```

4 clftp.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <fstream>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include "iosafe.h"

#define USAGE "Usage: clftp server-port server-hostname file-to-transfer filename-in-server"
#define BUF_SIZE 100000

using namespace std;

void inline sys_error_handle(const char* system_call)
{
    cerr << "Error: function:" << system_call << " errno:" << errno << ".\n";
    exit(1);
}

/*
 * call the socket
 */
int call_socket(char *hostname, unsigned short port_num)
{
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;
    memset(&sa, 0, sizeof(sa));

    if ((hp = gethostbyname(hostname)) == NULL)
    {
        errno= h_errno;
        sys_error_handle("gethostbyname");
    }

    sa.sin_family = hp->h_addrtype;
    memcpy((char*)&sa.sin_addr, hp->h_addr, hp->h_length);
    sa.sin_port = htons((u_short)port_num);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        sys_error_handle("socket");
    }

    if (connect(s, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    {
        sys_error_handle("connect");
    }

    return s;
}

/*
 * get the max file size from server and check if the file size is valid
```

```
 */
bool check_size(int src, char* buf, struct stat &statbuf)
{
    try
    {
        safe_read(src, buf, 4);
    }
    catch(exception& e)
    {
        sys_error_handle(e.what());
    }
    unsigned int maxSize = *(unsigned int*)buf;


    if (maxSize < statbuf.st_size)
    {
        return false;
    }
    return true;
}

int main(int argc, char* argv[])
{
    if (argc != 5)
    {
        cout << USAGE << endl;
        exit(1);
    }

    int s_port = atoi(argv[1]);
    char * s_hostname = argv[2];
    char * file_to_transfer = argv[3];
    char * file_in_server = argv[4];

    if (s_port < 1 || s_port > 65535 || (ifstream(file_to_transfer) == 0))
    {
        cout << USAGE << endl;
        exit(1);
    }

    //call the socket
    int s;
    s = call_socket(s_hostname, s_port);

    //create buffer
    char * buf = new char[BUF_SIZE];

    //to know and check file size
    struct stat statbuf;
    if (lstat(file_to_transfer, &statbuf) != 0)
    {
        sys_error_handle("lstat");
    }
    if(S_ISDIR(statbuf.st_mode))
    {
        cout << USAGE << endl;
        exit(1);
    }

    if (check_size(s, buf, statbuf) == false)
    {
        cout << "Transmission failed: too big file" << endl;
        send_size(s, buf, -1);
        delete[] buf;
        return 0;
    }

    unsigned int size = statbuf.st_size;
```

```
128
129         int f = open(file_to_transfer, 'R');
130         if (f < 0)
131         {
132             sys_error_handle("open");
133         }
134         try
135         {
136             send_size(s, buf, size); // sent file size
137             send_size(s, buf, string(file_in_server).size()); // send file in server length
138             send_name(s, buf, file_in_server, string(file_in_server).size()); // senr file in server name
139             safe_send(f, s, buf, BUF_SIZE, size);
140         }
141         catch(exception& e)
142         {
143             sys_error_handle(e.what());
144         }
145         delete[] buf;
146         if (close(f)==-1)
147         {
148             sys_error_handle("close");
149         }
150         return 0;
151     }
152     ;
```

5 iosafe.h

```
/*
 * iosafe.h
 *
 *  Created on: 12 jun 2015
 *      Author: roigreenberg
 */

#ifndef IOSAFE_H_
#define IOSAFE_H_

/*
 * iosafe.cpp
 *
 *  Created on: 12 jun 2015
 *      Author: roigreenberg
 */

#include <iostream>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <fstream>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>

using namespace std;

class SystemError: public exception
{
    char const* error_message;
public:
    SystemError(char const* error_name)
    {
        error_message = error_name;
    }
    virtual const char* what() const throw()
    {
        return error_message;
    }
};

/*
 * this function read or write until it finish all the requested bytes
 */
int io_safe(ssize_t (*f)(int, void*, size_t), int src, char *buf, unsigned int bufSize);

/*
 * helper for read
 */
ssize_t safe_read(int src, void* buf, size_t size);

/*
 * helper for write
 */
ssize_t safe_write(int dst, void* buf, size_t size);
```

```
60
61  /*
62   * send ALL the data from dst to src
63   */
64  int safe_send(int src, int dst, char* buf, unsigned int bufSize, size_t size);
65
66  /*
67   * sent int to dst
68   */
69  int send_size(int dst, char  *buf , unsigned int size);
70
71  /*
72   * sent name to dst
73   */
74  int send_name(int dst, char  *buf , char * name, unsigned int size);
75
76
77
78
79  #endif /* IOSAFE_H_ */
```

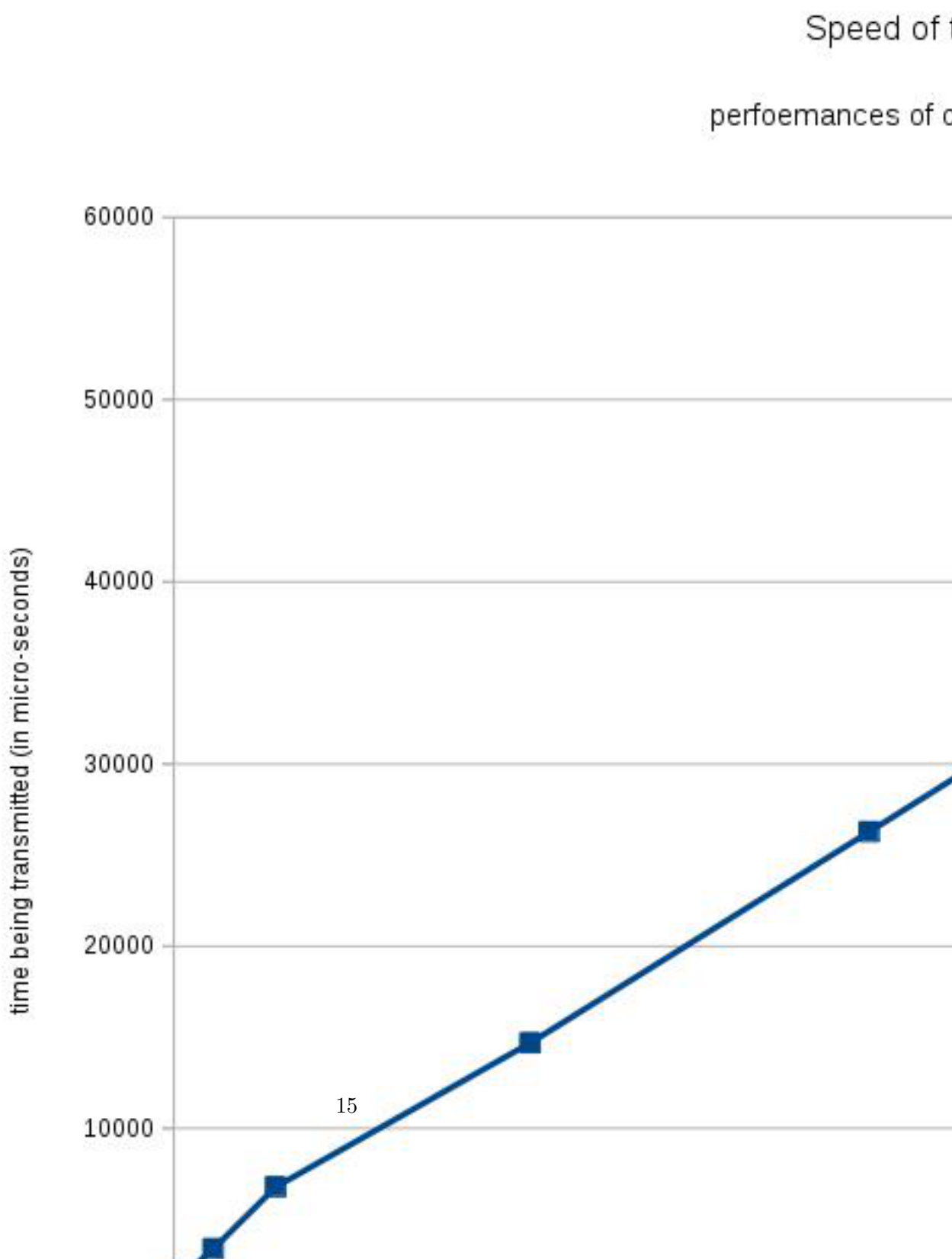
6 iosafe.cpp

```cpp
/*
 * iosafe.cpp
 *
 *  Created on: 12 jun 2015
 *      Author: roigreenberg
 */

#include <iostream>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <fstream>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include "iosafe.h"

using namespace std;


/*
 * this function read or write until it finish all the requested bytes
 */
int io_safe(ssize_t (*f)(int, void*, size_t), int src, char *buf, unsigned int bufSize)
{
    unsigned int count =0;
    int tmp = 0;

    while (count < bufSize)
    {
        if ((tmp = f(src, buf + count, bufSize - count)) > 0)
        {
            count += tmp;
        }
        else if (tmp < 0)
        {
            return -1;
        }

    }
    return count;
}


/*
 * helper for read
 */
ssize_t safe_read(int src, void* buf, size_t size)
{
    return read(src, buf, size);
}

/*
 * helper for write
 */
ssize_t safe_write(int dst, void* buf, size_t size)
```

```
60  {
61      return write(dst, buf, size);
62  }
63
64  /*
65   * send ALL the data from dst to src
66   */
67  int safe_send(int src, int dst, char* buf, unsigned int bufSize, size_t size)
68  {
69
70      int rcount = 0;
71      int wcount = 0;
72      unsigned int count = 0;
73      unsigned int left = size;
74
75      while (count < size)
76      {
77          if (left < bufSize)
78          {
79              bufSize = left;
80          }
81          rcount = io_safe(&safe_read, src, buf, bufSize);//errors?
82          if (rcount == -1)
83          {
84              throw SystemError("read");
85          }
86          wcount = io_safe(&safe_write, dst, buf, rcount);
87
88          if (wcount == -1)
89          {
90              throw SystemError("write");
91          }
92          count += wcount;
93          left -= wcount    ;
94      }
95
96      return count;
97  }
98
99  /*
100  * sent int to dst
101  */
102 int send_size(int dst, char  *buf , unsigned int size)
103 {
104     int n;
105     unsigned char * p_size = (unsigned char * )&size;
106     memcpy(buf, p_size, 4);
107     if ((n = safe_write(dst, buf, 4)) != 4)
108     {
109         throw SystemError("write");
110     }
111     return n;
112 }
113
114 /*
115  * sent name to dst
116  */
117 int send_name(int dst, char  *buf , char * name, unsigned int size)
118 {
119     unsigned int n;
120     memcpy(buf, name, size);
121     if ((n = safe_write(dst, buf, size)) != size)
122     {
123         throw SystemError("write");
124     }
125     for (size_t i = 0; i < size; i++)
126     {
127         buf[i] = '\0';
```

```
128         }
129         return n;
130     }
```

7 performance.jpg



8 srftp.cpp

```cpp
#include <iostream>
#include <fstream>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <pthread.h>
#include "iosafe.h"

#define BUF_SIZE 10000
#define USAGE "Usage: srftp server-port max-file-size"

using namespace std;

int s_port;
int max_file_size;

void inline sys_error_handle(const char* system_call)
{
    cerr<<"Error: function:" << system_call << " errno:" << errno << ".\n";
    pthread_exit(NULL);
}

int establish(unsigned short port_num)
{
    char myname[HOST_NAME_MAX + 1];
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;
    memset(&sa, 0, sizeof(struct sockaddr_in));


    if(gethostname(myname, HOST_NAME_MAX) == -1)
    {
        sys_error_handle("gethostname");
    }
    hp = gethostbyname(myname);
    if (hp == NULL)
    {
        errno= h_errno;
        sys_error_handle("gethostbyname");
    }
    sa.sin_family = hp->h_addrtype;

    sa.sin_port = htons((int)port_num);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        sys_error_handle("socket");
    }
    if (bind(s, (struct sockaddr *) &sa, sizeof(sa)) < 0)
    {
        sys_error_handle("bind");
```

```
60      }
61      if (listen(s, 5) < 0)
62      {
63          sys_error_handle("listen");
64      }
65      return (s);
66  }
67
68  int get_connection(int s)
69  {
70      int ns;
71      if ((ns = accept(s, NULL, NULL)) < 0)
72      {
73          sys_error_handle("accept");
74      }
75      return ns;
76  }
77
78
79  int client(int ns)
80  {
81      char* buf = new char[BUF_SIZE];
82      try
83      {
84          send_size(ns, buf, max_file_size);
85          int i;
86          // read file size
87          if ((i = safe_read(ns, buf, 4)) < 0)
88          {
89              sys_error_handle("read");
90          }
91          if (*(int*)buf == -1)
92          {
93              delete[] buf;
94              close(ns);
95              return 0;
96          }
97          unsigned int size = *(unsigned int*)buf;
98
99          // read file name length
100         if ((i = safe_read(ns, buf, 4)) < 0)
101         {
102             sys_error_handle("read");
103         }
104
105         unsigned int nameSize = *(unsigned int*)buf;
106
107         //read file name
108         char * filename = new char[nameSize+1];
109         if ((i = safe_read(ns, filename, nameSize)) < 0)
110         {
111             sys_error_handle("read");
112         }
113         filename[nameSize] = '\0'; //make sure file name end properly
114
115         int f;
116
117         ofstream of (filename);
118         of.close();
119
120         if ((f = open (filename, O_WRONLY)) < 0)
121         {
122             delete[] filename;
123             delete[] buf;
124             close(ns);
125             sys_error_handle("open");
126         }
127         //write data to file
```

```
128             safe_send(ns, f, buf, BUF_SIZE, size);
129             delete[] filename;
130             delete[] buf;
131             if (close(f)==-1 || close(ns)==-1)
132             {
133                 sys_error_handle("close");
134             }
135         }
136         catch (exception& e)
137         {
138             sys_error_handle(e.what());
139         }
140         return 0;
141     }
142     /*
143      * helper function for the client thread
144      * convert the ns to int
145      */
146     void* client_thread(void* ns)
147     {
148         int s = *(int*)ns;
149         delete (int*)ns;
150         client(s);
151         return NULL;
152     }
153
154     int main(int argc, char* argv[])
155     {
156         int ns;
157
158         if (argc != 3)
159         {
160             cout << USAGE << endl;
161             exit(1);
162         }
163
164         s_port = atoi(argv[1]);
165         const char * max_file = argv[2];
166         max_file_size = atoi(max_file);
167
168         if (s_port < 1 || s_port > 65535 || max_file_size < 0)
169         {
170             cout << USAGE << endl;
171             exit(1);
172         }
173
174         int s = establish(s_port);
175         while (true)
176         {
177             ns = get_connection(s);
178             pthread_t client;
179             if (pthread_create(&client, NULL, client_thread, (void*)new int(ns))!=0)
180             {
181                 sys_error_handle("pthread_create");
182             }
183             pthread_detach(client);
184         }
185         return s;
186     }
```