Spring 2016

EE/CSCI 450
Section 30560R

Programming Assignment #2

TCP Client/Server

Due 11:45PM 4/29/2016 (electronic submission only)

This spec is private (i.e., only for students who took or are taking EECS 450 at USC). You do not have permissions to display this spec at a public place (such as a github). You also do not have permissions to display the code you write to implementation this spec at a public place since your code can be used to reverse engineer the spec easily. (If a prospective employer asks you to post your code, please tell them that you do not have permissions to do so; but you can send them a private copy.)

Assignment Description

You will write two programs, a tserver and a tclient. The server (i.e., tserver) creates a stream socket (i.e., TCP) in the Internet domain bound to a port number (specified as a commandline argument when you start your server), receives requests from a client (i.e., tclient), produces results based on the content of requests, and sends the results back to the client on the same connection.

For this exercise, there are three types of messages you must implement:

- a request to get the file type of a file on the server (FILETYPE_REQ)
- a request to get the checksum of a file on the server (CHECKSUM_REQ)
- a request to download a file from the server (DOWNLOAD_REQ)

All messages have the same structure: a 1-byte MessageType field, a 4-byte (unsigned 32-bit integer) DataLength field, and a variable-length data field.

Byte Pos	Name	Description
0	MessageType	0xea (FILETYPE_REQ): file-type request 0xe9 (FILETYPE_RSP): successful file-type response 0xe8 (FILETYPE_ERR): failed file-type response 0xca (CHECKSUM_REQ): file checksum request 0xc9 (CHECKSUM_RSP): successful checksum response 0xc8 (CHECKSUM_ERR): failed checksum response 0xaa (DOWNLOAD_REQ): download file request 0xa9 (DOWNLOAD_RSP): successful download response 0xa8 (DOWNLOAD_ERR): failed download response 0x51 (UNKNOWN_FAIL): catch-all failure response
1-4	DataLength	Length of the Data field below (in network byte order).
5+	Data	Binary data byte stream. Meaning of this field depends on the MessageType field.

The server is normally in a sleeping state. When it receives a message from a client, it expects the message to be in the format described above. Here is the what the server must do when it gets a message of a certain type:

FILETYPE_REQ : The Data field is a filename (not null-terminated). The server should call popen() to execute a "file" command with "filename" as the argument (i.e., as if the command, "/usr/ucb/file filename" is executed on the server machine), read the output of the command, convert each tab character to a space character, and send it back to the client in a FILETYPE_RSP message. If filename is invalid, the server must send back a 5-byte long FILETYPE_ERR message.

CHECKSUM_REQ : The Data field is a 4-byte offset, followed by a 4-byte length, followed by a filename (not null-terminated). The server must compute a MD5 checksum for the data bytes of the specified file in the specified range and send the checksum value back to the client in a CHECKSUM_RSP message. If the message is too short, or the offset, the length, or the filename part of the Data is invalid or if it cannot open the specified file, the server must send back a 5-byte long CHECKSUM_ERR message.

DOWNLOAD_REQ : The Data field is a 4-byte offset, followed by a 4-byte length, followed by a filename (not null-terminated). The server must send the data bytes of the specified file in the specified range back to the client in a DOWNLOAD_RSP message. If the message is too short, or the offset, the length, or the filename part of the Data is invalid or if it cannot open the specified file, the server must send back a 5-byte long DOWNLOAD_ERR message.

? : If the server receives anything unrecognizable, it must send back a 5-byte long UNKNOWN_FAIL message.

Here are some additional requirements:

filename : If a filename is specified in a message, it must not be empty and it must contain only valid characters. Valid characters in a filename include numbers, uppercase or lowercase letters, dashes and underscores, periods and commas, and the plus symbol. All other characters are invalid (including the space character, the tab character, international character, etc.)

offset : An offset is a 32-bit unsigned integer specifed in the network byte order format. The value of offset must be $\geqslant 0$ and $\leqslant 2,147,483,647$ (or 0x7fffffff in hex). For a given file, if offset is larger or equal to the size of the file, it is considered an invalid offset.

length : A length is a 32-bit integer (signed) specifed in the network byte order format. If the value of length is negative, it means that you must send all the remaining data in the specified file starting with the specified offset. For a given file, if length is non-negative and offset+length is larger than the size of the file (equal is okay), it is considered an invalid length.

Extra Credit - Datagram Sockets

You can get up to 25% extra credit points if you also support datagram sockets (i.e., UDP). Basically, instead of having the client and server communicating over TCP sockets, you need to get the same thing to work over UDP sockets using a window-based protocol. There are a few more twists.

For the server, you need to support a loss model where you pretend that UDP packets are dropped. We will simulate a loss model by reading bits from a loss model file. Ever time you are ready to send an UDP packet from the server to the client, you read a bit from the loss model file. If the bit is a one, you send the UDP packet. If the bit is a zero, you don't send the UDP packet (and pretend that the packet was lost somewhere in the middle of the Internet). You must not retransmit a packet immediately; you can only retransmit a loss packet after a timeout interval has expired.

The maximum allowed UDP packet size is 4,096 bytes. Therefore, if your server has a long message to send to the client, it needs to be broken up into multiple UDP packets. What you need to do is to break a message above into 4,088 bytes long frames and add 8 bytes at the beginning of each frame as an UDP header. The first 4 bytes must be a sequence number (i.e., a frame number) in network byte order and the next 4 bytes is reserved for your protocol. You may design a longer header, but 8 bytes is the minimum. To make the packet format uniform, a UDP packet from the client to the server must follow the same format.

The following are left for you to design:

- Your design must use a window-based protocol. This means that you cannot simply implement a stop-and-wait protocol since its window size is one. You do not have to design for congestion control.
- You need to decide how to handle server retransmission. For example, you can ACK every packet. Alternatively, you can send a NACK to request a retransmission of a packet with a certain frame number.
- Please note that the client does not perform retransmission. If the request packet from the client never reached the server, the client would have no way to know that the server never got the message and the client would hang. This behavior would be fine.
- Please note that the client does not use a timeout mechanism. Therefore, it can only send something after it has received a packet from the server. The client also does not know the window size.
- You need to decide how the server can figure out when to free up buffers that it no longer needs.
- You need to decide what new MessageType code to use for any new message that you introduce and the meaning of the Data field in these messages.

You must document your design in the README file included with your submission.

Commandline Syntax & Program Output

The commandline syntax for the tserver and tclient is given below. The syntax is:

```
tserver [-d] [-t seconds] port

tclient [hostname:]port filetype filename
tclient [hostname:]port checksum [-o offset] [-l length] filename
tclient [hostname:]port download [-o offset] [-l length] filename [saveasfilename]
```

Square bracketed items are optional. You must follow the UNIX convention that commandline options can come in any order. (Note: a commandline option is a commandline argument that begins with a - character in a commandline syntax specification.) Unless otherwise specified, output of your program must go to stdout and error messages must go to stderr.

For tserver, if the -t commandline option is specified, seconds is the number of seconds it takes for the tserver to auto-shutdown and it must be ≥ 5. If -t is not specified, your server must auto-shutdown 300 seconds after it starts. If the -d commandline option is specified, it puts the server in debug mode. In the debug mode, the server must print the content of every message it received and sent to stdout. Please see debug mode below for more details.

For tclient, if a hostname is not specified, you must connect to "localhost". If -o is not specified, you must use an offset of zero. If -l is not specified, you must use a length of (-1), which is 0xffffffff for a 32-bit integer. If -o is specified, offset must be > 0. If -l is specified, length must be > 0.

The requests should have the message structure described above. tclient sends one request to the server, waits for a response, prints the result, and terminates itself.

For every response tclient receives, it must print one line of output. Here are the information about the required output:

FILETYPE_RSP : Check every byte in the Data field and make sure that every byte is ≤ 0x7f. If it is, you must print the entire Data field in one line to stdout. If it is not, you must print "Invalid characters detected in a FILETYPE_RSP message.\n" to stdout.

CHECKSUM_RSP : Since an MD5 checksum must be exactly 16 bytes long, if the DataLength field of the message is not 16, you must print "Invalid DataLength detected in a CHECKSUM_RSP message.\n" to stdout. Otherwise, you must print the hexstring representation of the Data field in one line to stdout. In this case, your output must contain exactly 32 hex characters followed by a "\n".

DOWNLOAD_RSP : After you have read the first 5 of this message and DataLength is > 0, you must save the Data part of the message into a file. If saveasfilename is specified in the commandline, you must check to see if the file already exists. If the file does not exists, you must write the Data part of the message into the file specified by saveasfilename.

If the file specified by saveasfilename already exists, you must print "File saveasfilename already exists, would you like to overwrite it? [yes/no](n) " (and replace saveasfilename with the last commandline argument). You must not print a "\n" and put the cursor right after your prompt to the user. The "(n)" in the message means that the default answer is "no". If the first letter of the user's response is "y", you must go ahead and overwrite the existing file. If the first letter of the user's response is anything else, you must print "Download canceled per user's request.\n".

If saveasfilename is not specified in the commandline, the file you need to create will have the same name as what you are downloading, i.e., you need to get it from filename. Since filename may be a file system path, you must search for the last "/" character and what comes after that is the file name you need to use. In this case, the file you need to create must be created in the current working directory (which is the output of the Unix command "pwd"). The rest of the logic is the same as the logic above for saving a file into a user-specified file name.

When you have successfully write Data into FILE, you must print "Downloaded data have been successfully written into 'FILE' (MD5=...)\n" where FILE is either saveasfilename (when applicable) or the actual file name you wrote into and ... is the MD5 checksum of Data in hexstring format.

When you download the Data portion of the message, you must use an algorithm similar to this one to read from the socket incrementally:

```
unsigned char buf[4096];
int remaining=DataLength;

while (remaining > 0) {
    int bytes_to_read=min(remaining,4096);
    int bytes_read=read(socket, buf, bytes_to_read);
    printf(".");
    fflush(stdout);
    remaining -= bytes_read;
};
```

The above code reads at most 4,096 bytes from the socket and print a dot to stdout to show progress.

? : If the client receives anything else, it must print "Unexpected message of type MSGTYPE received.\n", and you must replace MSGTYPE with the name of the MessageType if it's recognized. If the MessageType is not recognized, you must replace MSGTYPE with the hexstring representation of the MessageType you received.

Below are some examples of running the client (assuming you have a tserver running on nunki.usc.edu serving port 12345). A percent sign (%) at the beginning of a line indicates the UNIX command prompt.

```
% tclient nunki.usc.edu:12345 filetype /usr/bin
/usr/bin: directory

% tclient nunki.usc.edu:12345 filetype /usr/bin/ls
/usr/bin/ls: ELF 32-bit MSB executable SPARC Version 1, dynamically linked, stripped

% tclient nunki.usc.edu:12345 filetype /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
/home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png: PNG image data

% tclient nunki.usc.edu:12345 checksum /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
29fa9fc4d619fe576b57972b8c5fb9e4

% tclient nunki.usc.edu:12345 download /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
...Downloaded data have been successfully written into 'logo-viterbi.png' (MD5=29fa9fc4d619fe576b57972b8c5fb9e4)

% tclient nunki.usc.edu:12345 checksum -o 1 /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
9dc0314a5213a3a9e67375af872e8463

% tclient nunki.usc.edu:12345 checksum -l 20 /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
0a19b5803ef78555d9c8554d5d747113

% tclient nunki.usc.edu:12345 checksum -o 1 -l 20 /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
2d33ec017797006baf52fa21c3c82b13

% tclient nunki.usc.edu:12345 filetype xyzzy
xyzzy:  cannot open: No such file or directory

% tclient nunki.usc.edu:12345 filetype /home/scf-22/csci551b/xyzzy2/x
/home/scf-22/csci551b/xyzzy2/x: cannot open: Permission denied

% tclient nunki.usc.edu:12345 filetype '%@#^'
FILETYPE_ERR received from the server

% tclient nunki.usc.edu:12345 checksum -o 9106 /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
CHECKSUM_ERR received from the server

% tclient nunki.usc.edu:12345 checksum -l 9107 /home/scf-22/csci551b/public/eecs450/pa2/logo-viterbi.png
CHECKSUM_ERR received from the server
```

If an error condition whose handling method has not been explicitly specified, please think about what would make sense and handle it appropriately and print out reasonable and useful error messages.

Extra Credit - Datagram Sockets

If you are supporting UDP for extra credit, you need to be able to indicate that you are running the server and the client in the UDP mode. The commandline syntax will then be:

```
tserver [-udp loss_model [-w window] [-r msinterval]] [-d] [-t seconds] port

tclient [hostname:]port filetype [-udp] filename
tclient [hostname:]port checksum [-udp] [-o offset] [-l length] filename
tclient [hostname:]port download [-udp] [-o offset] [-l length] filename [saveasfilename]
```

The loss_model names a binary file and you should read it one byte at a time. When you need a bit to determine if you need to throw a packet away or not, you should look at the left-most bit in the byte that has not been examined. Once you have use all 8 bits in a byte, you should read the next byte. If you have reached the end of the loss model file, you should start from the beginning of the file again.

You are required to use a window-based protocol to provide reliability where window is the size of the window (which must be ⩾ 1) and msinterval is the timeout interval in milliseconds (which must be ⩾ 1 and ⩽ 5000). If the -w commandline option is not specified, you must use a window size of 3. If the -r commandline option is not specified, you must use a timeout interval of 250.

Compiling and Linking

Please use a single Makefile so that when the grader simply enters:

```
make tclient
```

an executable named tclient is created. If the grader simply enters:

```
make tserver
```

an executable named tserver is created. Please make sure that your submission conforms to other general compilation requirements.

If you do man -s 3socket socket, it says:

```
cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]
#include <sys/types.h>
#include <sys/socket.h>
```

The #include is what you need to put into your source code. The -lsocket -lnsl is what you need to include when you link your modules together to create your executables.

Which Port Number Should You Use?

On nunki.usc.edu (and any machine), you cannot have two servers listening on the same port. To minimize the chances that two students are using the same port numbers, here is how you should choose a port number for your server. Let X be the last 3 digits of your USC student ID number. Your port number should be ⩾ 20000+10X and ⩽ 20009+10X. For example, if the last 3 digits of your USC student ID number is 012, your port number should be ⩾ 20120 and ⩽ 20129. Since you have a range of 10 port numbers to choose from, if one of them is not working, you should just move on to the next one. If none of them are working, you can run "netstat -an | grep PORT" (where PORT is a number) to check if PORT is actually being used.

Server Debug Mode

When you put the server in the debug mode, you must print the content of every message it received and sent to stdout. Below is a table of what the server should print to stdout when it recevied a particular type of message:

FILETYPE_REQ	:	FILETYPE_REQ received with DataLength = ???, Data = '...'\n
CHECKSUM_REQ	:	CHECKSUM_REQ received with DataLength = ???, offset = ???, length = ???, filename = '...'\n
DOWNLOAD_REQ	:	DOWNLOAD_REQ received with DataLength = ???, offset = ???, length = ???, filename = '...'\n
?	:	Message with MessageType = 0x?? received. Ignored.\n

Please replace "???" and "..." with the information related to the message you sent.

Below is a table of what the server should print to stdout when it sends a particular type of message:

FILETYPE_RSP	:	FILETYPE_RSP sent with DataLength = ???, Data = '...'\n
FILETYPE_ERR	:	FILETYPE_ERR sent with DataLength = ???\n
CHECKSUM_RSP	:	CHECKSUM_RSP sent with DataLength = ???, checksum = ...\n
CHECKSUM_ERR	:	CHECKSUM_ERR sent with DataLength = ???\n
DOWNLOAD_RSP	:	DOWNLOAD_RSP sent with DataLength = ???\n
DOWNLOAD_ERR	:	DOWNLOAD_ERR sent with DataLength = ???\n
UNKNOWN_FAIL	:	UNKNOWN_FAIL sent with DataLength = ???\n

Please replace "???" and "..." with the information related to the message you sent. For checksum, replace "..." with the hexstring representation of the Data you sent.

In the debug mode, when the server auto-shuts down, it must print "??? seconds timer has expired. Sever has auto-shutdown.\n" where ??? is the auto-shutdown interval.

Extra Credit - Datagram Sockets

If you are supporting UDP for extra credit, you need to take care of the following for the server in debug mode:

- Instead of printing one line after every message, you need to print one line after every packet. You must include the sequence number and other UDP header information in your print out. Let's take UNKNOWN_FAIL as an example. You should now print "[###/???] UNKNOWN_FAIL sent with DataLength = ???\n" if the packet is sent, where ### within the square brackets is the sequence number and ??? within the square brackets is the other UDP header information. On the other hand, if the packet is dropped, you should print "[###/???/DROPPED] UNKNOWN_FAIL sent with DataLength = ???\n" instead.

- For the additional message types you will receive from the client, you must print debugging message to stdout providing similar information as above. Please make sure you document these in your README file.

Hexstring Representation

To display an array of bytes (binary data), you must convert each byte of data into exactly two hexadecimal digits and print them out. For example, if buf is an array of unsigned characters and you want to print the first 8 bytes of buf to stdout in hexstring representation, you can do the following:

```
for (int i=0; i < 8; i++) {
    printf("%02x", buf[i]);
}
```

Auto-shutdown the Server

To auto-shutdown your server, you need to do the following:

- Use the alarm() call to delilver a SIGALRM signal to your program.
- Catch SIGALRM with a signal handler. To specify a signal handler, you should use sigset().
- Inside the signal handler, call shutdown() to shutdown the server's master socket, call close() to close the master socket, then call exit() to quit your program. To learn more about the master socket, please see the Sockets Programming section below. (Please understand that this is not really a graceful way to quit your program! But it's fine for now.)

popen()

A very convenient way of programmatically executing a program and reading its output is to use popen(). The following code segment executes /usr/bin/date, reads the output produced by /usr/bin/date and prints it out to stdout.

```
char cmd[80];
FILE *pfp=NULL;

snprintf(cmd, sizeof(cmd), "/usr/bin/date");
if ((pfp=(FILE*)popen(cmd, "r")) == NULL) {
    fprintf(stderr, "Cannot execute '%s'.\n", cmd);
} else {
    while (fgets(buf, sizeof(buf), pfp) != NULL) {
        fprintf(stdout, "%s", buf);
    }
    pclose(pfp);
}
```

Please do man popen to see the man pages for popen().

MD5 Checksum Calculation

To calculate the MD5 checksum of a file, you can use the OpenSSL library. Please follow the openssl instruction to set things up on nunki.usc.edu so you can use OpenSSL there. After you have set things up according to the openssl instruction, you can do "man MD5" on nunki.usc.edu to see a list of MD5 related functions and instructions on how to use them. Here's part of what "man MD5" would show:

```
SYNOPSIS
```

```
#include <openssl/md5.h>

unsigned char *MD5(const unsigned char *d, unsigned long n, unsigned char *md);

void MD5_Init(MD5_CTX *c);
void MD5_Update(MD5_CTX *c, const void *data, unsigned long len);
void MD5_Final(unsigned char *md, MD5_CTX *c);
```

There are two ways to calculate MD5 checksum for an array of unsigned characters. The first way is to do it in one shot by calling MD5(buf, n, md5_sum). where buf is a fixed size buffer (e.g., unsigned char buf[256]) and md5_sum is declared as unsigned char md5_sum[MD5_DIGEST_LENGTH]. When MD5(buf, n, md5_sum) returns, md5_sum will contain the MD5 checksum for the first n bytes of buf. You should only do this if the array size of buf is small.

If you need to compute the MD5 checksum for a large number of bytes, you should do it incrementally. Here's the basic code:

```
#include <openssl/md5.h>

MD5_CTX c;
unsigned char buf[4096]; /* buffer to keep data */
unsigned char md5_sum[MD5_DIGEST_LENGTH]; /* output checksum */

MD5_Init(&c);
while (!done) {
    int n = FillBuf(buf, sizeof(buf)); /* n = number of bytes in buf */
    MD5_Update(&c, buf, n);
}
MD5_Final(md5_sum, &c);
```

The idea of FillBuf(buf, sizeof(buf)) above is that it will fill buf up to the size of the buffer and will return the number of bytes that was put into buf. For this assignment, the server will fill the buffer using data read from the file and the client will fill the buffer using data read from its connection to the server.

Sockets Programming

One good place to go to learn about sockets programming online is Beejs Guide to Network Programming. There is a ton of information there. Make sure you checkout the page on client-server programming with TCP and UDP. Try out the sample code there (both client and server) and use it as the starting place for your client and server programs.

Grading Guidelines

[LG: section added 4/20/2016]
The grading guidelines has been made available (currently does not contain information about extra credit stuff). Please run the scripts in the guidelines on nunki.usc.edu. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

Please note that although the grader will follow the grading guidelines to grade, the grader may use a different set of numeric values and commandline arguments.

The grading guidelines is the only grading procedure we will use to grade your program. No other grading procedure will be used. (We may make minor changes if we discover bugs in the script or things that we forgot to test.) It is strongly recommended that you run your code through the scripts in the grading guidelines.

Miscellaneous Requirements

- Your server should be robust. It should never crash. It should always generate a response if it can (that is what the UNKNOWN_FAIL message type is for).

- Clean up! If you are using multiple processes you should not leave zombie processes behind. Use "ps x" to see what processes you have left lying around. You can use "kill -9 PID" to kill a process with a given "process ID".

Some Programming Hints

- Let's say that you want to kill your server process and restart it immediately. But your

- Let's say that you want to kill your server process and restart it immediately. But your call to bind() fails. Try the following and see if it helps:

```
int my_socket=0, reuse_addr=1;

... [create my_socket]...

if (setsockopt(my_socket, SOL_SOCKET, SO_REUSEADDR,
        (void*)(&reuse_addr), sizeof(int)) == -1) {
    /* report error */
}
```

- If you are having trouble with figuring out how to set a timer to shutdown the server or for the server to break out of the accept() call, here's some information that may be helpful.

[Last updated Wed Apr 20 2016]   [Please see copyright regarding copying.]